shortcut

Your Short Cut to Knowledge

Debugging Linux Systems

Sreekrishnan Venkateswaran

PRENTICE HALL
PEARSON EDUCATION

INFORMIT.COM/PH

What This Mini-Book Will Cover

Visit us on the Web: www.informit.com/ph

Pearson Education, Inc.
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax: (617) 671-3447

ISBN-13: 978-0-136-12354-5
ISBN-10: 0-136-12354-6

First release, November 2009

Product Description

Debugging Linux Systems discusses the main tools available today to debug 2.6 Linux Kernels. We start by exploring the seemingly esoteric operations of the Kernel Debugger (KDB), Kernel GNU DeBugger (KGDB), the plain GNU DeBugger (GDB), and JTAG debuggers. We then investigate Kernel Probes, a feature that lets you intrude into a kernel function and extract debug information or apply a medicated patch. Analyzing a crash dump can yield clues for postmortem analysis of kernel crashes or hangs, so we take a look at *Kdump*, a serviceability tool that collects a system dump after spawning a new kernel. Profiling points you to code regions that burn more CPU cycles, so we learn to use the *OProfile* kernel profiler and the *gprof* application profiler to sense the presence of code bottlenecks. Because tracing provides insight into behavioral problems that manifest during interactions between different code modules, we delve into the Linux Trace Toolkit, a system designed for high-volume trace capture.

The section "Debugging Embedded Linux" takes a tour of the I/O interfaces commonly found on embedded hardware, such as flash memory, serial port, PCMCIA, Secure Digital media, USB, RTC, audio, video, touch screen, and Bluetooth, and provides pointers to debug the associated device drivers. We also pick up some board-level debugging skills with the help of a case study. The section "Debugging Network Throughput" takes you through some device driver design issues and protocol implementation characteristics that can affect the horsepower of your network interface card. We end the shortcut by examining several options available in the kernel configuration menu that can emit valuable debug information.

Kernel Version

This shortcut is generally up to date as of the 2.6.25/26 kernel versions. Most code listings in this shortcut were tested on a 2.6.23 kernel. If you are using a later version, look at Linux websites such as lwn.net to learn about the kernel changes made since the above releases.

Book Website

I've set up a website at elinuxdd.com to provide updates, errata, and other information related to *Essential Linux Device Drivers* and this digital shortcut.

Conventions Used

Source code, function names, and shell commands, are written like this. The shell prompt used is **`bash>`**. Filenames are written in italics, *like this*. Italics are also used to introduce new terms.

Some sections modify original kernel source files while implementing code examples. To clearly point out the changes, newly inserted code lines are prefixed with +, and any deleted code lines with -.

Sometimes, for simplicity, this shortcut uses generic references. So if the text points you to the *arch/your-arch/* directory, it should be translated, for example, to *arch/x86/* if you are compiling the kernel for the x86 architecture.

The → symbol is sometimes inserted between command or kernel output to attach explanations.

Some sections refer you to user-space scripts or configuration files. The exact names and locations of such files might, however, vary according to the Linux distribution you use.

Dedication

This digital shortcut, like Essential Linux Device Drivers, *is dedicated to the ten million visually challenged citizens of India. All author proceeds will go to their cause.*

Investing time in logic design and software engineering before code development and staring hard at the code after development can minimize or even eliminate bugs. But because that is easier said than done, and because we are all humans, developers need debugging tools. In this shortcut, let's explore a variety of methods to debug kernel code. The techniques described here have been stitched together from different parts of the book *Essential Linux Device Drivers* (Prentice Hall Open Source Software Development Series, ISBN 978-0-132-39655-4).

Reliability, Availability, Serviceability

Many systems, especially mission critical ones, have a need for reliability, availability, and serviceability (RAS). The Linux RAS effort has resulted in the development of several powerful tools. Exercisers such as the Linux Test Project (LTP) measure the reliability and robustness of your kernel port. CPU hotplugging and the Linux High Availability (HA) project can be seen in the context of availability. Kernel debuggers, Kprobes, Kdump, EDAC, and the Linux Trace Toolkit (LTT) come under the ambit of serviceability. The line dividing these classifications is sometimes thin; you can use any or a combination of these methods to suit your debugging needs. For example, output from a kernel profiler such as *OProfile* can be used either to search for potential code bottlenecks (reliability) or to debug a field problem (serviceability).

Kernel Debuggers

The instruction-level Kernel DeBugger (kdb) and the source-level Kernel GNU DeBugger (kgdb) are the two main Linux kernel debuggers. Whether to include a debugger as part of the stock kernel has been an oft-debated point in kernel mailing lists, but a lightweight version of kgdb has finally been integrated with the mainline kernel starting with the 2.6.26 release. Even if you prefer to stay away from the seemingly esoteric operation of kernel debuggers, you can glean information about

kernel panics and peek at kernel variables via the plain GNU DeBugger (gdb). JTAG debuggers use hardware-assisted debugging and are powerful but expensive.

Kernel debuggers make kernel internals more transparent. You can single-step through instructions, disassemble instructions, display and modify kernel variables, and look at stack traces. In this section, let's learn the basics of kernel debuggers with the help of some examples.

Entering a Debugger

You can enter a kernel debugger in multiple ways. One way is to pass command-line arguments that ask the kernel to enter the debugger during boot. Another way is via software or hardware breakpoints. A *breakpoint* is an address where you want execution stopped and control transferred to the debugger. A software breakpoint replaces the instruction at that address with something else that causes an exception. You can set software breakpoints either using debugger commands or by inserting them into your code. For x86-based systems, you can set a software breakpoint in your kernel source code as follows:

```
asm(" int $3");
```

Alternatively, you can invoke the `BREAKPOINT` macro, which translates to the appropriate architecture-dependent instruction.

You can use hardware breakpoints in place of software breakpoints if the instruction where you need to stop is in flash memory, where it cannot be replaced by the debugger. A hardware breakpoint needs processor support. The corresponding address needs to be added to a debug register. You can only have as many hardware breakpoints as the number of debug registers supported by the processor.

You can also ask a debugger to set a *watchpoint* on a variable. The debugger stops execution whenever an instruction modifies data at the watchpoint address.

Yet another common method to enter a debugger is by pressing an attention key, but this won't work in many instances. If your code is sitting in a tight loop after disabling interrupts, the kernel will not get a chance to process the attention key and enter the debugger. For example, you can't enter the debugger via an attention key if your code does something like this:

```
unsigned long flags;

local_irq_save(flags);
while (1) continue;
local_irq_restore(flags);
```

When control is transferred to the debugger, you can start your analysis using various debugger commands.

Kernel Debugger (kdb)

Kdb is an instruction-level debugger used for debugging kernel code and device drivers. Before you can use it, you need to patch your kernel sources with kdb support and recompile the kernel. (Refer to the section "Downloads" for information on downloading kdb patches.) The main advantage of kdb is that it's easy to set up, because you don't need an additional machine to do the debugging (unlike kgdb). The main disadvantage is that you need to correlate your sources with disassembled code (again, unlike kgdb).

Let's wet our toes in kdb with the help of an example. Here's the crime scene: You have modified a kernel serial driver to work with your x86-based hardware, but the driver isn't working, and you want kdb to help nab the culprit.

Let's start our search for fingerprints by setting a breakpoint at the serial driver `open()` entry point. Remember, because kdb is not a source-level debugger, you need to open your sources and try to match the instructions with your C code. Let's list the source snippet in question:

drivers/serial/myserial.c:

```
static int rs_open(struct tty_struct *tty, struct file *filp)
{
  struct async_struct *info;

  /* ... */
  retval = get_async_struct(line, &info);
  if (retval) return(retval);
  tty->driver_data = info;
  /* Point A */

  /* ... */
}
```

Press the Pause key and enter kdb. Let's find out how the disassembled `rs_open()` looks. The debug sessions shown here attach explanations using the → symbol.

```
Entering kdb (current=0xc03f6000, pid 0) on processor 0 due to
Keyboard Entry

kdb> id rs_open           → Disassemble rs_open
0xc01cce00 rs_open:       sub $0x1c, %esp
0xc01cce03 rs_open+0x03:  mov $ffffffed, %ecx
...
0xc01cce4b rs_open+0x4b:  call 0xc01ccca0, get_async_struct
...
0xc01cce56 rs_open+0x56:  mov 0xc(%esp,1), %eax
0xc01cce5a rs_open+0x5a:  mov %eax, 0x9a4(%ebx)
...
```

Point A in the source code is a good place to attach a breakpoint because you can peek at both the `tty` structure and the `info` structure to see what's going on.

Looking side by side at the source and the disassembly, `rs_open+0x5a` corresponds to Point A. Note that correlation is easier if the kernel is compiled without optimization flags.

Set a breakpoint at `rs_open+0x5a` (which is address `0xc01cce5a`) and continue execution by exiting the debugger:

```
kbd> bp rs_open+0x5a    → Set breakpoint
kbd> go                 → Continue execution
```

Now you need to get the kernel to call `rs_open()`to hit the breakpoint. To trigger this, execute an appropriate user-space program. In this case, echo some characters to the corresponding serial port *(/dev/ttySX)*:

```
bash> echo "Anjali loves kerala monsoons" > /dev/ttySX
```

This results in the invocation of `rs_open()`. The breakpoint gets hit, and kdb assumes control:

```
Entering kdb on processor 0 due to Breakpoint @ 0xc01cce5a
kdb>
```

Let's now find out the contents of the `info` structure. If you look at the disassembly, one instruction before the breakpoint (`rs_open+0x56`), you see that the `EAX` register contains the address of the `info` structure. Let's look at the register contents:

```
kbd> r              → Dump register contents
eax = 0xcf1ae680 ebx = 0xce03b000 ecx = 0x00000000
...
```

So, `0xcf1ae680` is the address of the `info` structure. Dump its contents using the `md` command:

```
kbd> md 0xcf1ae680  → Memory dump
0xcf1ae680 00005301 0000ABC 00000000 10000400
...
```

To make sense of this dump, let's look at the corresponding structure definition. `info` is defined as `struct async_struct` in *include/linux/serialP.h* as follows:

```
struct async_struct {
  int            magic;  /* Magic Number */
  unsigned long  port;   /* I/O Port */
  int            hub6;
  /* ... */
};
```

If you match the dump with the definition, `0x5301` is the magic number and `0xABC` is the I/O port. Well, isn't this interesting! `0xABC` doesn't look like a valid port. If you have done enough serial port debugging, you know that the I/O port base addresses and IRQs are configured in *include/asm-x86/serial.h* for x86-based hardware. Change the port definition to the correct value, recompile the kernel, and continue your testing!

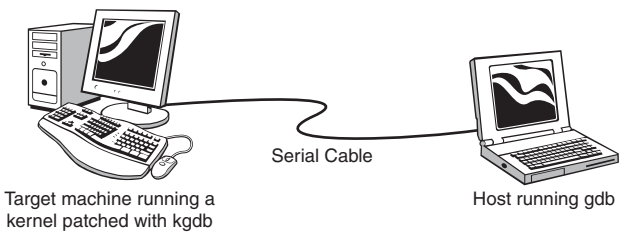
Kernel GNU Debugger (kgdb)

Kgdb is a source-level debugger. It is easier to use than kdb because you don't need to spend time correlating assembly code with your sources. However, it's more difficult to set up because an additional machine is needed to front-end the debugging.

You have to use gdb in tandem with kgdb to step through kernel code. gdb runs on the host machine, whereas the kgdb-enabled kernel runs on the target hardware. The host and the target are connected via a serial null-modem cable, as shown in Figure 1.1.[1]

[1] *You can also launch kgdb debug sessions over Ethernet.*

FIGURE 1.1
Kgdb setup



You have to inform the kernel about the identity and baud rate of the serial port via command-line arguments. Depending on the bootloader used, add the following kernel arguments to either *syslinux.cfg, lilo.conf*, or *grub.conf*:

```
kgdbwait kgdb8250=X,115200
```

`kgdbwait` asks the kernel to wait until a connection is established with the host-side gdb, `X` is the serial port connected to the host, and `115200` is the baud rate used for communication.

Now configure the same baud rate on the host side:

```
bash> stty speed 115200 < /dev/ttySX
```

If your host computer is a laptop that does not have a serial port, you can use a USB-to-serial converter for the debug session. In that case, instead of */dev/ttySX*, use the */dev/ttyUSBX* node created by the usbserial driver.

Let's learn kgdb basics using the example of a buggy kernel module. Modules are easier to debug because the entire kernel need not be recompiled after making code changes, but remember to compile your module with the `-g` option to generate symbolic information. Because modules are dynamically loaded, the debugger needs to be informed about the symbolic information that the module contains. Listing 1.1 contains a buggy `trojan_function()`. Assume that it's defined in *drivers/char/my_module.c.*

LISTING 1.1 Buggy Function

```
char buffer;

int
trojan_function()
{
  int *my_variable = 0xAB, i;

  /* ... */
  Point A:
  i = *my_variable; /* Kernel Panic: my_variable points
                       to bad memory */
  return(i);
}
```

Insert *my_module.ko* on the target and look inside */sys/module/my_module/sections/* to decipher ELF (*Executable and Linking Format*) section addresses.[2] The `.text` section in ELF files contains code, `.data` contains initialized variables, `.rodata` contains initialized read-only variables such as strings, and `.bss` contains variables that are not initialized during startup. The addresses of these sections are available in the form of the files *.text*, *.data*, *.rodata*, and *.bss* in */sys/module/my_module/sections/* if you enable `CONFIG_KALLSYMS` during kernel configuration. To obtain the code section address, for instance, do this:

```
bash> cat /sys/module/my_module/sections/.text
0xe091a060
```

More module load information is available from */proc/modules* and */proc/kallsyms.*

After you have the section addresses, invoke gdb on the host-side machine:

```
bash> gdb vmlinux                → vmlinux is the uncompressed kernel

(gdb) target remote /dev/ttySX → Connect to the target
```

[2] *If you still use a 2.4 kernel, get the section addresses using the –m option to insmod instead:*

```
bash> insmod my_module.o -m
Using /lib/modules/2.x.y/kernel/drivers/char/my_module.o
Sections:           Size      Address   Align
.this           00000060  e091a000  2**2
.text           00001ec0  e091a060  2**4
...
.rodata           0000004c  e091d1fc  2**2
.data           00000048  e091d260  2**5
.bss            000000e4  e091d2c0  2**5
...
```

Because you passed `kgdbwait` as a kernel command-line argument, gdb gets control when the kernel boots on the target. Now inform gdb about the preceding section addresses using the `add-symbol-file` command:

```
(gdb) add-symbol-file drivers/char/mymodule.ko 0xe091a060
      -s .rodata 0xe091d1fc -s .data 0xe091d260 -s .bss 0xe091d2c0

add symbol table from file "drivers/char/my_module.ko" at
        .text_addr = 0xe091a060
        .rodata_addr = 0xe091d1fc
        .data_addr = 0xe091d260
        .bss_addr = 0xe091d2c0
(y or n) y

Reading symbols from drivers/char/mymodule.ko ...done.
```

To debug the kernel panic, let's set a breakpoint at `trojan_function()`:

```
(gdb) b trojan_function          → Set breakpoint
(gdb) c                          → Continue execution
```

When kgdb hits the breakpoint, let's look at the stack trace, single-step until Point A, and display the value of `my_variable`:

```
(gdb) bt                         → Back (stack) trace
#0 trojan_function () at my_module.c :124
#1 0xe091a108 in my_parent_function (my_var1=438, my_var2=0xe091d288)
..
```

```
(gdb) step
(gdb) step                          → Continue to single-step up to
                                       Point A
(gdb) p my_variable
$0 = 0
```

There is an obvious bug here. `my_variable` points to `NULL` because `trojan_function()` forgot to allocate memory for it. Let's just allocate the memory using kgdb, circumvent the kernel crash, and continue testing:

```
(gdb) p &buffer                     → Print address of buffer
$1 = 0xe091a100 ""
(gdb) set my_variable=0xe091a100 → my_variable = &buffer
(gdb) c                             → Continue your testing
```

Note

Kgdb ports are available for several architectures such as x86, ARM, and PowerPC. When you use kgdb to debug a target embedded device (instead of the PC shown in Figure 1.1), the gdb front-end that you run on your host system needs to be compiled to work with your target platform. For example, to debug a device driver developed for an ARM-based embedded device from your x86-based host development system, you need to use the appropriately generated gdb, often named arm-linux-gdb. The exact name depends on the distribution you use.

GNU Debugger (gdb)

As previously mentioned, you can use plain gdb to gather some kernel debug information. However, you can't step through kernel code, set breakpoints, or modify kernel variables. Let's use gdb to debug the kernel panic caused by the buggy function in Listing 1.1, but assume this time that `trojan_function()` is compiled as part of the kernel and not as a module, because you can't easily peek inside modules using gdb.

This is part of the "oops" message generated when `trojan_function()` is executed:

```
Unable to handle kernel NULL pointer dereference at
virtual address 000000ab
 ...
 eax: f7571de0   ebx: ffffe000   ecx: f6c78000   edx: f98df870
 ...
 Stack: c019d731 00000000
        ...
        bffffbe8 c0108fab
 Call Trace:    [<c019d731>] [<c013b8ac>] [<c0108fab>]
 ...
```

Copy this cryptic "oops" message to *oops.txt* and use the *ksymoops* utility to obtain more verbose output. You might need to hand-copy the message if the system is hung:

```
bash> ksymoops oops.txt
Code;  c019d710 <trojan_function+0/10>
00000000 <_EIP>:
```

```
Code;  c019d710 <trojan_function+0/10>   <=====
   0:   a1 ab 00 00 00            mov    0xab,%eax   <=====
Code;  c019d715 <trojan_function+5/10>
   5:   c3                        ret
```

2.6 kernels emit "oops" output that can be used as is without the need of decoding using *ksymoops* if you enable `CONFIG_KALLSYMS` during kernel configuration.

Looking at the preceding dump, the "oops" has occurred inside `trojan_-function()`. Let's use gdb to obtain more information. In the following invocation, *vmlinux* is the uncompressed kernel image, and */proc/kcore* is the kernel address space:

```
bash> gdb vmlinux /proc/kcore

(gdb) p xtime      → Test the waters by printing a kernel variable
$0 = 1113173755
```

Repeated access to the same variable will not yield refreshed values due to gdb's cached access. You can force a fresh access by rereading the core file using gdb's `core-file` command. Let's now look at the disassembly of `trojan_function()`:

```
(gdb) x/2i trojan_function       ???? Disassemble trojan_function
0xc019d710 <trojan_function>:    mov 0xab, %eax
0xc019d715 <trojan_function+5>:  ret
```

`trojan_function()` looks laconic when seen in assembly due to compiler optimizations. It's effectively copying the contents of address `0xab` to the `EAX` register, which holds the return value from

functions on x86-based systems. But `0xab` does not look like a valid kernel address! Fix the bug by allocating valid memory space to `my_-variable`, recompile, and continue your testing.

JTAG Debuggers

JTAG debuggers use hardware-assist to debug code. You need specialized monitor hardware[3] and a front-end user interface (some JTAG debuggers use gdb as the front-end) to step through code. JTAG can also be used for purposes other than debugging, such as burning code onto onboard flash memory. JTAG connectors are common on development boards but are usually not part of production units.

JTAG debuggers usually connect to target hardware via serial port, USB, or Ethernet. With Ethernet, you can remotely access the JTAG debugger, and therefore the target board, even if the board itself does not possess a network interface.

Figure 1.2 shows a JTAG-based remote debugging session in action. The JTAG debugger used in this scenario supports a gdb front end. The development host and the JTAG hardware are connected to an Ethernet LAN. The debug serial port on the target hardware is connected to the serial port on the JTAG box. Figure 1.2 achieves remote debugging on the Linux development host using five terminal sessions. Terminal 1 runs gdb, which connects to the JTAG box over the network using telnet:

[3] *Some JTAG debuggers work with several processor architectures if you suitably replace the probe that connects the debugger to the target board.*

```
(gdb) target remote 10.1.1.2:1001 ?→ 10.1.1.2 is the IP address of
                                     the JTAG hardware. 1001 is the
                                     JTAG TCP port that listens to
                                     incoming gdb connections
```

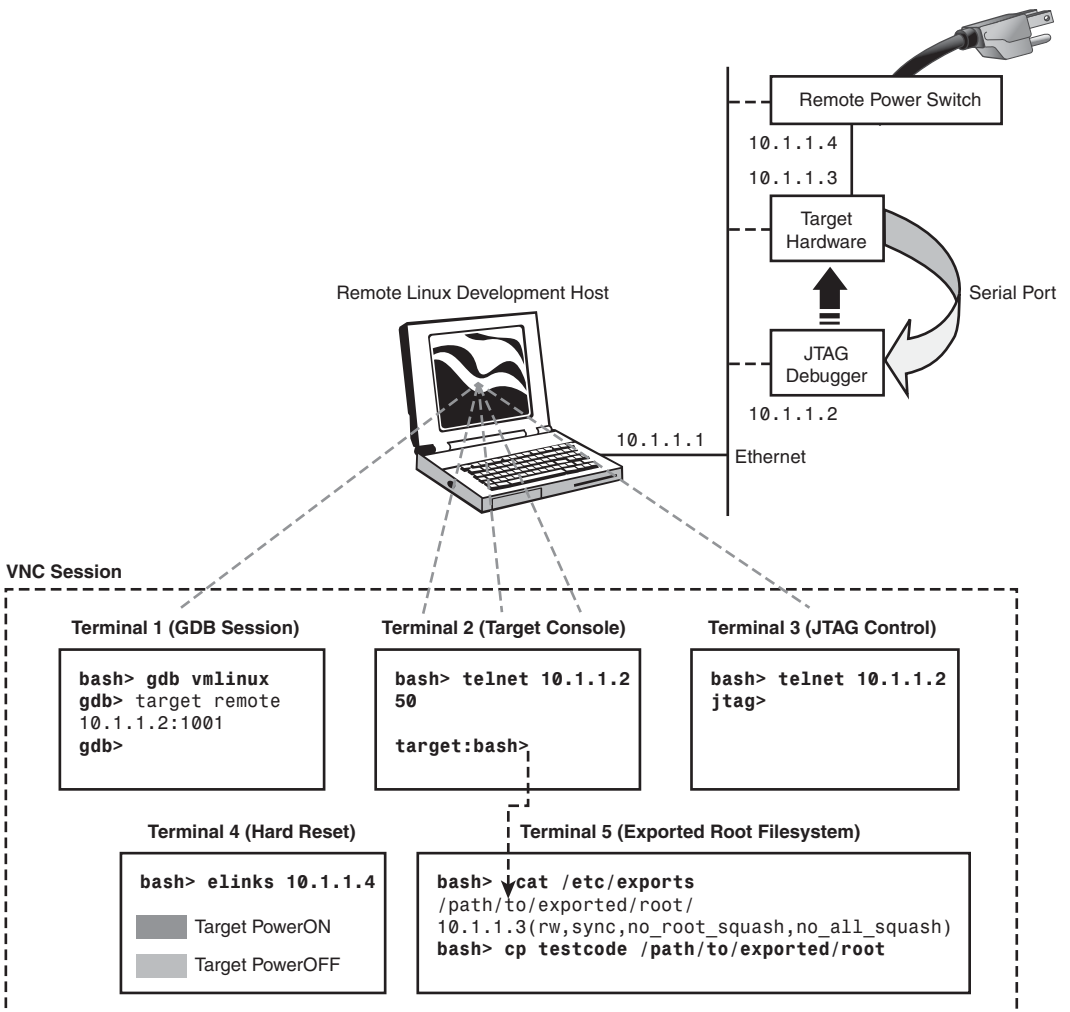
To debug boot portions of the kernel, for example, set a gdb breakpoint at `start_kernel()`. (You can find its address from *System.map*, which is generated in the root of your source tree when you build the kernel.)

Terminal 2 attaches a serial console to the target. A telnet client running on Terminal 2 connects to a prespecified TCP port on the JTAG box, which is configured (using Terminal 3) to tunnel data arriving via its serial port:

```
bash> telnet 10.1.1.2 50          → 10.1.1.2 is the IP address of
                                     the JTAG hardware. 50 is the
                                     JTAG TCP port that tunnels data
                                     arriving via its serial port
```

This is equivalent to running an emulator such as *minicom* after directly connecting the target's debug serial port to the host (instead of to the JTAG box, as shown in Figure 1.2), but that'll constrain the host to be physically adjacent to the target.

FIGURE 1.2
An example JTAG-based remote debug setup



Terminal 3 telnets to the JTAG box and offers debugger-specific semantics. You can use it, for example, to do the following:

- Pull a JTAG definition script over TFTP from the host and execute it during JTAG boot. A JTAG definition script usually initializes the processor, clock registers, chip select registers, and memory banks. After this is done, the JTAG hardware is ready to download code onto the target and execute it. The JTAG manufacturer usually provides definition files for all supported platforms, so you are likely to have a close starting point for your board.
- Download your bootloader, kernel, or stand-alone code from the host over TFTP, to flash memory, or RAM on the target. File formats such as ELF and binary are usually supported by JTAG debuggers.
- Single-step code, set breakpoints, examine registers, and dump memory regions.
- Reset the target.

JTAG debugging can be flaky at times, so if you are debugging remotely, it might be a good idea to power the target via a remote power control switch, as shown in Figure 1.2. That way, you can hard-reset the target from the host using a web browser, as shown in Terminal 4. You can also choose to power the JTAG hardware via a remote power switch. That enables you to test run a bootloader directly from flash without the intervention of JTAG and its definition files.

If the target board possesses a network interface, it can mount its root filesystem over NFS from the development host. Terminal 5 on the host operates locally on the exported root filesystem.[4]

[4] *You might have more such terminals depending on your debug scenario. If you use an oscilloscope that has remote display capabilities, for example, you can operate it via a web browser on another terminal.*

If your team is scattered geographically, run Terminals 1 through 5 within an environment such as Virtual Network Computing (VNC). If VNC is not already part of your distribution, download it from www.realvnc.com. With such a setup, you can debug the electrons on your remote board from the comfort of your home! Some JTAG vendors provide a sophisticated integrated development environment[5] that encompasses all the functionalities previously detailed, so you don't need to manage VNC terminal sessions if you're using one of those.

During hardware bring up, when you are porting your bootloader or other stand-alone code to the target, it's a good idea to first generate an ELF image and debug it from RAM before running it from flash. Remember, however, to eliminate bootloader initializations that duplicate the ones performed by the JTAG definition script.

A key advantage of JTAG debuggers is that you can use a single tool to debug the different pieces that constitute your firmware solution. So, you can use the same debugger to debug the BIOS, bootloader, base kernel, device driver modules, and user-space applications, at source level.

Downloads

You can download kdb patches for the x86 and IA64 architectures from http://oss.sgi.com/projects/kdb. Each supported kernel version needs two patches: a common one and an architecture-dependent one.

The home page for the kgdb project is http://kgdb.sourceforge.net. The website also has documentation on configuring and using kgdb.

[5] *Although JTAG hardware is independent of the target operating system, the front-end interface is likely to have OS dependencies.*

If your Linux distribution does not already contain gdb, you can obtain it from www.gnu.org/software/gdb/gdb.html.

Kernel Probes

Kernel probes can intrude into a kernel function and extract debug information or apply a medicated patch. It's a useful addition to your debugging repertoire for investigating inexplicable behavior at a customer site, especially when you don't have the luxury of rebooting the system. Linux supports a generic form of kernel probes called *Kprobes* and two specialized variants, *Jprobes* and *return probes.*

Kprobes

Kprobes can save you the trouble of building and booting a debug kernel by providing capabilities to dynamically dump kernel data structures or insert code into a running kernel. You can, for example, add a few `printk`s on-the-fly inside the scheduler without recompiling the kernel. You can even patch a bug on a Mars rover without rebooting it.

To insert a kprobe inside a kernel function, follow these steps:

1. Turn on `CONFIG_KPROBES` (*Instrumentation Support* → *Kprobes*) in the kernel configuration menu.
2. Implement a kernel module that registers a kprobe at the instruction of interest. You need to register a *pre-handler* that Kprobes will run just before executing the probed instruction and a *post-handler* that Kprobes will run after executing the probed instruction. You can also supply a *fault-handler* that will run if a fault is detected while executing the pre- or post-handlers (because you don't want to "oops" due to a debugging bug!).

When a kprobe is registered, it saves the probed instruction and replaces it with an instruction that generates a breakpoint (`int 0x03` on x86-based systems). When the breakpoint is hit, the kernel generates a *die* notification. Kprobes inserts itself into the die notifier chain, so it gets notified about the breakpoint hit.

When notified, Kprobes executes the registered pre-handler. Next, it steps through a copy of the probed instruction. It executes a copy instead of swapping the probed instruction with the breakpoint instruction for reasons of SMP consistency. Finally, it runs the post-handler. The pre- and post-handler windows are the hooks offered to the Kprobes user to inject debug code. The handlers can be registered and unregistered on-the-fly, so serviceability is not merely static at compile time but programmable during runtime.

Let's learn to use Kprobes with the help of an example. Consider the code snippet in Listing 1.2, which is a kernel thread that adds `npages` number of pages to the free memory pool, whenever a `SIGUSR1` signal is delivered to it. Most of the logic has been scissored out of the listing because it's not relevant. Assume that you are at a customer site to debug a problem reported with this code. You notice bad things whenever `npages` crosses `10`, so you want to apply a runtime patch that limits it to `10`.

LISTING 1.2 Problem Code (mydrv.c)

```
int npages=0;
EXPORT_SYMBOL(npages);

static int memwalkd(void *unused)
{
  long curr_pfn = (64*1024*1024 >> PAGE_SHIFT);
```

```
  struct page *curr_page;
  /* ... */

  daemonize("memwalkd"); /* kernel thread */

  sigfillset(&current->blocked);
  allow_signal(SIGUSR1);

  for (;;) {
    /* Dequeue a signal if it's pending */
    if (signal_pending(current)) {
      sig = dequeue_signal(current, &current->blocked, &info);
      /* Point A */
      /* Free npages pages when SIGUSR1 is received */
      if (sig == SIGUSR1) {
      /* Point B */
        /* Problem manifests when npages crosses 10 in the following
           loop. Let's apply run time medication here via Kprobes */
        for (i=0; i < npages; i++, curr_pfn++) {
          /* ... */
        }
      }
      /* ... */
  }
  /* ... */
}
```

LISTING 1.3 Registering Kprobe Handlers

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/kprobes.h>
#include <linux/kallsyms.h>
#include <linux/sched.h>

extern int npages; /* Defined in Listing 1.2 */

/* Per-probe structure */
static struct kprobe bandaid;

/* Pre Handler: Invoked before running probed instruction */
int bandaid_pre(struct kprobe *p, struct pt_regs *regs)
{
  if (npages > 10) npages = 10;
  return 0;
}

/* Post Handler: Invoked after running probed instruction */
void bandaid_post(struct kprobe *p, struct pt_regs *regs,
                  unsigned long flags)
{
  /* Nothing to do */
}
```

```
/* Fault Handler: Invoked if the pre/post-handlers
   encounter a fault */
int bandaid_fault(struct kprobe *p, struct pt_regs *regs,
                  int trapnr)
{
  return 0;
}

int init_module(void)
{
  int retval;

  /* Fill the kprobe structure */
  bandaid.pre_handler   = bandaid_pre;
  bandaid.post_handler  = bandaid_post;
  bandaid.fault_handler = bandaid_fault;

  /* Arrive at the target address as explained */
  bandaid.addr = (kprobe_opcode_t*)
                 kallsyms_lookup_name("memwalkd") + 0xaa;

  if (!bandaid.addr) {
    printk("Bad Probe Point\n");
    return -1;
  }
```

```
  /* Register the kprobe */
  if ((retval = register_kprobe(&bandaid)) < 0) {
    printk("register_kprobe error, return value=%d\n",
            retval);
    return -1;
  }
  return 0;
}

void module_cleanup(void)
{
  unregister_kprobe(&bandaid);
}

MODULE_LICENSE("GPL"); /* You can't link the Kprobes API
                          unless your user module is GPL'ed */
```

Listing 1.3 uses Kprobes to insert a patch at `kallsyms_lookup_name("memwalkd") + 0xaa`, which limits `npages` to `10`. To figure out how to arrive at this probe address, take another look at Listing 1.2. You want the patch to be inserted at Point B. To calculate the kernel address at Point B, disassemble the contents of *mydrv.ko* using *objdump*:

```
bash> objdump -D mydrv.ko

mydrv.ko:     file format elf32-i386

Disassembly of section .text:
```

```
00000000 <memwalkd>:
   0:     55                      push   %ebp
   1:     bd 00 40 00 00          mov    $0x4000,%ebp
   6:     57                      push   %edi
   7:     56                      push   %esi
   8:     53                      push   %ebx
   9:     bb 00 f0 ff ff          mov    $0xffff000,%ebx
   e:     81 ec 90 00 00 00       sub    $0x90,%esp
  ...
  ...
  7a:     83 f8 0a                cmp    $0xa,%eax   → Point A
  7d:     74 2b                   je     aa <memwalkd+0xaa>
  7f:     83 f8 09                cmp    $0x9,%eax
  82:     75 cc                   jne    50 <memwalkd+0x50>
  ...
  a9:     c3                      ret
  aa:     a1 00 00 00 00          mov    0x0,%eax    → Point B
  af:     85 c0                   test   %eax,%eax
  b1:     0f 8e b5 00 00 00       jle    16c <memwalkd+0x16c>
  b7:     81 fd 7b f6 00 00       cmp    $0xf67b,%ebp
  ...
  fa:     a1 00 00 00 00          mov    0x0,%eax
```

Note

You have to use an architecture-specific objdump if you're cross-compiling for a different processor platform. You need something like arm-linux-objdump if you disassemble a binary cross-compiled for an ARM-based target device. Pass the -S option to objdump to mix source code with the disassembled output:

```
bash> arm-linux-objdump -d -S mydrv.ko
```

If you try and match the C code in Listing 1.2 with its preceding disassembled dump, you can associate Point A and Point B with the shown kernel addresses. `kallsyms_lookup_name()`[6] locates the address of `memwalkd()`, and `0xaa` is the offset where Point B resides, so apply the kprobe at `kallsyms_lookup_name("memwalkd") + 0xaa`.

After you register the kprobe, memwalkd() equivalently looks like this:

```
static int memwalkd(void *unused)
{
  /* ...*/
  for (;;) {
    /* ... */
    /* Point A */
    /* Free npages pages when SIGUSR1 is received */
    if (sig == SIGUSR1) {
    /* Point B */
      if (npages > 10) npages = 10; /* The medicated patch! */
```

[6] *You have to enable CONFIG_KALLSYMS during kernel configuration to obtain the services of this function.*

```
        for (i=0; i < npages; i++, curr_pfn++) {
          /* ... */
        }
      }
      /* ... */
    }
    /* ... */
}
```

Whenever `npages` is assigned a value greater than `10`, the kprobed patch pulls it back to `10`, thus stepping around the problem.

In the next two sections, let's look at a couple of helper facilities that make it easier to use Kprobes during function entry and exit.

Jprobes

A *jprobe* is a specialized kprobe. It eases the work of adding a probe when the point of investigation is at the entry to a kernel function. The jprobe handler has the same prototype as the probed function. It's invoked with the same argument list as the probed function, so you can easily access the function arguments from the jprobe handler. If you use Kprobes rather than Jprobes, imagine the hassles your probe handler needs to undergo, wading through the dark alleys of the function stack to extract function arguments! And this code that delves into the stack to elicit argument values has to be heavily function-specific, not to mention being architecture-dependent and unportable.

To learn how to use Jprobes, let's revert to an example. Assume that you're debugging a network device driver (that is built as part of the kernel) by looking at the `printk()` messages it's generating. The driver is emitting crucial values in octal (base 8), but to your horror, the driver writer has introduced a typo in the print format string by coding `%0` rather than `%o`. So, all you can see are messages such as this:

```
Number of Free Receive buffers = %0.
```

Jprobes to the rescue. You can fix this in a few seconds, without recompiling or rebooting the kernel. First, take a look at `printk()` defined in *kernel/printk.c*:

```
asmlinkage int printk(const char *fmt, ...)
{
  va_list args;
  int r;

  va_start(args, fmt);
  r = vprintk(fmt, args);
  va_end(args);
  return r;
}
```

Let's add a simple jprobe at the entry to `printk()` and transform every `%0` into `%o`. Listing 1.4 does this job. Note that the jprobe handler needs to have the same prototype as `printk()`. Both functions are marked with the `asmlinkage` tag that asks them to expect arguments from the stack, rather than from CPU registers.

LISTING 1.4 Registering Jprobe Handlers

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/kprobes.h>
#include <linux/kallsyms.h>

/* Jprobe the entrance to printk */
asmlinkage int
jprintk(const char *fmt, ...)
{
  for (; *fmt; ++fmt) {
    if ((*fmt=='%')&&(*(fmt+1) == 'O')) *(char *)(fmt+1) = 'o';
  }
  jprobe_return();
  return 0;
}

/* Per-probe structure */
static struct jprobe jprobe_eg = {
  .entry = (kprobe_opcode_t *) jprintk
};

int
init_module(void)
{
  int retval;
```

```
  jprobe_eg.kp.addr = (kprobe_opcode_t*)
                      kallsyms_lookup_name("printk");

  if (!jprobe_eg.kp.addr) {
    printk("Bad probe point\n");
    return -1;
  }

  /* Register the Jprobe */
  if ((retval = register_jprobe(&jprobe_eg) < 0)) {
    printk("register_jprobe error, return value=%d\n",
            retval);
    return -1;
  }
  printk("Jprobe registered.\n");
  return 0;
}

void
module_cleanup(void)
{
  unregister_jprobe(&jprobe_eg);
}

MODULE_LICENSE("GPL");
```

When Listing 1.4 invokes `register_jprobes()` to register the jprobe, a kprobe is inserted at the beginning of `printk()`. When this probe is hit, Kprobes replaces the saved return address with that of the registered jprobe handler `jprintk()`. It then copies a portion of the stack and returns, thus passing control to `jprintk()` with `printk()`'s argument list. When `jprintk()` calls `jprobe_return()`, the original call state is restored, and `printk()` continues to execute normally.

When you insert this jprobe user module, the network driver no longer emits useless messages announcing `%0` buffers, rather it prints saner information such as this:

```
Number of Free Receive buffers = 12.
```

Return Probes

A *return probe* (or a *kretprobe* in Kprobes terminology) is another specialized Kprobes helper. It eases the work of inserting a kprobe when you need to probe a function's return point. If you use vanilla Kprobes to investigate return points, you might need to register them at multiple places because a function can return via multiple code paths. However, if you use return probes, you need to insert only one kretprobe, rather than register, say, 20 Kprobes to cover a function's 20 return paths.

The function `tty_open()` defined in *drivers/char/tty_io.c* has seven return paths. The successful path returns `0`, and others return error values such as `-ENXIO` and `-ENODEV`. A single kretprobe is sufficient to alert you about failures, irrespective of the associated code path. Listing 1.5 implements this kretprobe.

LISTING 1.5 Registering Return Probe Handlers

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/kprobes.h>
#include <linux/kallsyms.h>

/* kretprobe at exit from tty_open() */
static int
kret_tty_open(struct kretprobe_instance *kreti,
              struct pt_regs *regs)
{
  /* The EAX register contains the function return value
     on x86 systems */
  if ((int) regs->eax) {
    /* tty_open() failed. Announce the return code */
    printk("tty_open returned %d\n", (int)regs->eax);
  }
  return 0;
}

/* Per-probe structure */
static struct kretprobe kretprobe_eg = {
  .handler = (kretprobe_handler_t)kret_tty_open
};

int
init_module(void)
```

```
{
  int retval;

  kretprobe_eg.kp.addr = (kprobe_opcode_t*)
                         kallsyms_lookup_name("tty_open");

  if (!kretprobe_eg.kp.addr) {
    printk("Bad Probe Point\n");
    return -1;
  }

  /* Register the kretprobe */
  if ((retval = register_kretprobe(&kretprobe_eg) < 0)) {
    printk("register_kretprobe error, return value=%d\n",
            retval);
    return -1;
  }
  printk("kretprobe registered.\n");
  return 0;
}

void module_cleanup(void)
{
  unregister_kretprobe(&kretprobe_eg);
}

MODULE_LICENSE("GPL");
```

When Listing 1.5 invokes `register_kretprobes()`, a kprobe is internally inserted at the beginning of `tty_open()`. When this probe gets hit, this internal kprobe handler replaces the function return address with that of a special routine (called a *trampoline* in Kprobes terminology). Look at *arch/your-arch/kernel/kprobes.c* for the implementation of the trampoline.

When `tty_open()` returns via any of its seven return paths, control returns to the trampoline instead of the caller function. The trampoline invokes the kretprobe handler `kret_tty_open()`, registered by Listing 1.5, which prints the return value if `tty_open()` has not returned successfully.

Limitations

Kprobes has its limitations. Some of them are obvious. You won't, for example, see desired results if you insert a kprobe inside an inline function. And, of course, you can't probe Kprobes code.

Kprobes are more useful for applying probes inside the base kernel. If the subject code is part of a dynamically loadable module, you might as well rewrite and recompile your module rather than write and compile a new module to "kprobe" it. However, you might still want to use Kprobes if bringing down the module is not acceptable.

There are less-obvious limitations, too. Optimizations are done at compile time, whereas Kprobes are inserted during runtime. So, the effect of inserting instructions via Kprobes is not equivalent to adding code in the original source files. For example, the buggy code snippet

```
volatile int *integerp = 0xFF;
int integerd = *integerp;
```

is reduced by the compiler to

```
mov 0xff, %eax
```

So, you can't easily use Kprobes if you want to sneak in between those two lines of C code, allocate a word of memory, point `integerp` to the allocated word, and circumvent a kernel crash.

Note

SystemTap (http://sourceware.org/systemtap/) is a diagnostic tool that eases the use of Kprobes.

Looking at the Sources

The Kprobes implementation consists of a generic portion defined in *kernel/kprobes.c* (and *include/linux/kprobes.h*) and an architecture-dependent part that lives in *arch/your-arch/kernel/kprobes.c* (and *include/asm-your-arch/kprobes.h*).

Peek inside *Documentation/kprobes.txt* for further information about Kprobes, Jprobes, and Kretprobes.

Kexec and Kdump

Now that you have learned how to use Kprobes, let's continue and look at more facets of Linux RAS. *Kexec* and *kdump* are serviceability features introduced in the 2.6 kernel.

Kexec uses the image overlay philosophy of the UNIX `exec()` system call to spawn a new kernel over a running kernel without the overhead of boot firmware. This can save several seconds of

reboot time because boot firmware spends cycles walking buses and recognizing devices. The less the reboot latency, the less the system downtime; so, this was one of the main motivations for developing kexec. However, kexec's most popular user is kdump. Capturing a dump after a kernel crash is inherently unreliable because kernel code that accesses the dump device might be in an unstable state. Kdump circumvents this problem by collecting the dump after booting into a healthy kernel via kexec.

Kexec

Before you can kexec a kernel, you need to do some preparations:

1. Compile and boot into a kernel that has kexec support. For this, turn on `CONFIG_KEXEC` (*Processor Type and Features* → *Kexec System Call*) in the kernel configuration menu. This kernel is called the first kernel or the running kernel.
2. Prepare the kernel that is to be kexec-ed. This second kernel can be the same as the first kernel.
3. Download the *kexec-tools* package source tar ball from www.kernel.org/pub/linux/kernel/people/horms/kexec-tools/kexec-tools-testing.tar.gz. Build and produce the user-space tool called *kexec*.

The kexec tool built in Step 3 is invoked in two stages. The first stage loads the second kernel image into the buffers of the running kernel, whereas the second stage actually overlays the running kernel:

1. Load the second (overlay) kernel using the *kexec* command:

```
bash> kexec -l /path/to/kernelsources/arch/x86/boot/bzImage --
append="root=/dev/hdaX" --initrd=/boot/myinitrd.img
```

   bzImage is the second kernel, *hdaX* is the root device, and *myinitrd.img* is the initial root filesystem. The kernel implementation of this stage is mostly architecture-independent. At the heart of this stage is the `sys_kexec()` system call. The *kexec* command loads the new kernel image into the running kernel's buffers using the services of this system call.

2. Boot into the second kernel:

```
bash> kexec -e
...                    → Kernel boot up messages
```

   Kexec abruptly starts the new kernel without gracefully halting the operating system. To shut down prior to reboot, invoke kexec from the bottom of the *halt* script (usually */etc/rc.d/rc0.d/S01halt*) and invoke halt instead.

   The implementation of the second stage is architecture-dependent. The crux of this stage is a `reboot_code_buffer` that contains assembly code to put the new kernel in place to boot.

Kexec bypasses the initial kernel code that invokes the services of boot firmware and directly jumps to the protected mode entry point (for x86 processors). An important challenge to implement kexec is the interaction that happens between the kernel and the boot firmware (BIOS on x86-based systems, Openfirmware on POWER-based machines, and so on). On x86 systems, information such as the *e820* memory map passed to the kernel by the BIOS needs to be supplied to the kexec-ed kernel, too.

Kexec with Kdump

The kexec invocation semantics are somewhat special when it's used in tandem with kdump. In this case, kexec is required to automatically boot a new kernel when it encounters a kernel panic. If the running kernel crashes, the new kernel (called the capture kernel) is booted to reliably collect the dump. A typical call syntax is this:

```
bash> kexec -p /path/to/capture-kernel-sources/vmlinux
            --args-linux --append="root=/dev/hdaX irqpoll"
            --initrd=/boot/myinitrd.img
```

The `-p` option asks kexec to trigger a reboot when a kernel panic occurs. A *vmlinux* ELF kernel image is used as the capture kernel. Because *vmlinux* is a general ELF boot image and because kexec is theoretically OS-agnostic, you need to specify via the `--args-linux` option that the following arguments have to be interpreted in a Linux-specific manner. The capture kernel boots asynchronously during a kernel crash, so device drivers using shared interrupts may fatally express their unhappiness during boot. To be nice to such drivers, specify `irqpoll` in the command line passed to the capture kernel using `--append`.

To use kexec with kdump, you need some additional kernel configuration settings. The capture kernel requires access to kernel memory of the crashed kernel to generate a full dump, so the latter cannot just overwrite the former as was done by kexec in the non-kdump case. The running kernel needs to reserve a memory region to run the capture kernel. To mark this region:

- Boot the first kernel with the command-line argument `crashkernel=64M@16M` (or other suitable `size@start` values). Also include debug symbols in the kernel image by enabling `CONFIG_DEBUG_INFO` (*Kernel Hacking* → *Compile the Kernel with Debug Info*) in the configuration menu.

- While configuring the capture kernel, set `CONFIG_PHYSICAL_START` to the same `start` value assigned above (16M in this case). If you kexec into the capture kernel and peek inside */proc/meminfo*, you will find that `size` (64M in this case) is the total amount of physical memory that this kernel can see.

Now that you're comfortable with kexec and have mastered it from the perspective of a kdump user, let's delve into kdump and use it to analyze some real-world kernel crashes.

Kdump

An image of system memory captured after a kernel crash or hang is called a *crash dump.* Analyzing a crash dump can give valuable clues for postmortem analyses of kernel problems. However, obtaining a dump after a kernel crash is inherently unreliable because the storage driver responsible for logging data onto the dump device might be in an undefined state.

Until the advent of kdump, *Linux Kernel Crash Dump (LKCD)* was the popular mechanism to obtain and analyze dumps. LKCD uses a temporary dump device (such as the swap partition) to capture the dump. It then warm reboots back to a healthy state and copies the dump from the temporary device to a permanent location. A tool called *lcrash* is used to analyze the dump. The disadvantages with LKCD include the following:

- Even copying the dump to a temporary device might be unreliable on a crashed kernel.
- Dump device configuration is nontrivial.
- The reboot might be slow because swap space can be activated only after the dump has been safely saved away to a permanent location.

- LKCD is not part of the mainline kernel, so installing the proper patches for your kernel version is a hurdle.

Kdump is not burdened with these shortfalls. It eliminates indeterminism by collecting the dump after booting into a healthy kernel via kexec. Also, because memory state is preserved after a kexec reboot, the memory image can be accurately accessed from the capture kernel.

Let's first get the preliminary kdump setup out of the way:

1. Ask the running kernel to kexec into a capture kernel if it encounters a panic. The capture kernel should additionally have `CONFIG_HIMEM` and `CONFIG_CRASH_DUMP` turned on. (Both these options sit inside *Processor type and Features* in the kernel configuration menu.)
2. After the capture kernel boots, copy the collected dump information from `/proc/vmcore` (obtained by enabling `CONFIG_PROC_VMCORE` in the kernel configuration menu) to a file on your hard disk:

   ```
   bash> cp /proc/vmcore /dump/vmcore.dump
   ```

   You can also save other information such as the raw memory snapshot of the crashed kernel, via */dev/oldmem.*
3. Boot back into the first kernel. You are now ready to start dump analysis.

Let's use the collected dump file and the *crash* tool to analyze some example kernel crashes. Introduce this bug inside the interrupt handler of the RTC driver (*drivers/char/rtc.c*):

```
irqreturn_t rtc_interrupt(int irq, void *dev_id)
{
+  volatile int *integerp = 0xFF;
```

```
+  int integerd = *integerp;  /* Bad memory reference! */

   spin_lock(&rtc_lock);
   /* ... */
```

Trigger execution of the handler by enabling interrupts via the `hwclock` command:

```
bash> hwclock
...   → Kernel panic occurs when execution hits rtc_interrupt()
        causing kexec to boot into the capture kernel.
```

Save */proc/vmcore* to */dump/vmcore.dump*, reboot back into the first (crashed) kernel, and start analysis using the crash tool. In a real-world situation, of course, the dump might be captured at a customer site, whereas the analysis is done at a support center:

```
bash> crash /usr/src/linux/vmlinux /dump/vmcore.dump
crash 4.0-2.24
...
      KERNEL: /usr/src/linux/vmlinux
    DUMPFILE: /root/vmcore.dumpfile
        CPUS: 1
        DATE: Mon Nov 26 04:15:49 2007
      UPTIME: 00:17:22
LOAD AVERAGE: 0.82, 1.02, 0.87
       TASKS: 63
       ...
       PANIC: "Oops: 0000 [#1]" (check log for details)
crash>
```

Examine the stack trace to understand the cause of the crash:

```
crash> bt
PID: 0      TASK: c03a32e0  CPU: 0   COMMAND: "swapper"
 #0 [c0431eb8] crash_kexec at c013a8e7
 #1 [c0431f04] die at c0103a73
 #2 [c0431f44] do_page_fault at c0343381
 #3 [c0431f84] error_code (via page_fault) at c010312d
    EAX: 00000008  EBX: c59a5360  ECX: c03fbf90  EDX: 00000000
    EBP: 00000000
    DS:  007b      ESI: 00000000  ES:  007b      EDI: c03fbf90
    CS:  0060      EIP: f8a8c004  ERR: ffffffff  EFLAGS: 00010092
 #4 [c0431fb8] rtc_interrupt at f8a8c004
 #5 [c0431fc4] handle_IRQ_event at c013de51
 #6 [c0431fdc] __do_IRQ at c013df0f
```

The stack trace points the needle of suspicion at `rtc_interrupt()`. Let's disassemble the instructions near `rtc_interrupt()`:

```
crash> dis 0xf8a8c000 5
0xf8a8c000 <rtc_interrupt>:     push   %ebx
0xf8a8c001 <rtc_interrupt+1>:   sub    $0x4,%esp
0xf8a8c004 <rtc_interrupt+4>:   mov    0xff,%eax
0xf8a8c009 <rtc_interrupt+9>:   mov    $0xc03a6640,%eax
0xf8a8c00e <rtc_interrupt+14>:  call   0xc0342300 <_spin_lock>
```

The instruction at address `0xf8a8c004` is attempting to move the contents of the `EAX` register to address `0xff`, which is clearly the invalid deference that caused the crash. Fix this and build a new kernel.

If you use the `irq` command, you can figure out the identity of the interrupt that was in progress during the time of the crash. In this case, the output confirms that the RTC interrupt was indeed active:

```
crash> irq
    IRQ: 8
 STATUS: 1 (IRQ_INPROGRESS)
...
...
handler: f8a8c000  <rtc_interrupt>
            flags: 20000000  (SA_INTERRUPT)
             mask: 0
             name: f8a8c29d  "rtc"

crash> quit
bash>
```

Let's now shift gears and look at a case where the kernel freezes, rather than generate an "oops." Consider the following buggy driver `init()`routine:

```
static int __init
mydrv_init(void)
{
```

```
  spin_lock(&mydrv_wq.lock);  /* Usage before initialization! */
  spin_lock_init(&mydrv_wq.lock);

  /* ... */
}
```

The code is erroneously using a spinlock before initializing it. Effectively, the CPU spins forever trying to acquire the lock, and the kernel appears to hang. Let's debug this problem using kdump. In this case, there will be no auto-trigger because there is no panic, so force a crash dump by pressing the magic Sysrq key combination, Alt-Sysrq-c. You may need to enable Sysrq by writing a 1 to */proc/sys/kernel/sysrq*:

```
bash> echo 1 > /proc/sys/kernel/sysrq
bash> insmod mydrv.ko
```

This induces the kernel to hang inside `mydrv_init()`. Press the Alt-Sysrq-c key combination to trigger a crash dump:

```
Alt-Sysrq-c
...                    → Messages announcing that a crash dump
                         has been triggered
```

Save the dump to disk after kexec boots the capture kernel, boot back to the original kernel, and run crash on the saved dump:

```
bash> crash vmlinux vmcore.dump
      ...
      PANIC: "SysRq : Trigger a crashdump"
```

```
          PID: 2115
      COMMAND: "insmod"
         TASK: f7c57000  [THREAD_INFO: f6170000]
          CPU: 0
        STATE: TASK_RUNNING (SYSRQ)
crash>
```

Test the waters by checking the identity of the process that was running at the time of the crash. In this case, it was apparently *insmod* (of *mydrv.ko*):

```
crash> ps
    ...
    2171   2137   0  f6bb7000  IN   0.5   11728   5352  emacs-x
    2214      1   0  f6b5b000  IN   0.1    2732   1192  login
    2230   2214   0  f6bb0550  IN   0.1    4580   1528  bash
  > 2261   2230   0  c596f550  RU   0.0    1572    376  insmod
```

The stack trace doesn't yield much information other than telling you that a Sysrq key press was responsible for the dump:

```
crash> bt
PID: 2115   TASK: f7c57000  CPU: 0   COMMAND: "insmod"
 #0 [c0431e68] crash_kexec at c013a8e7
 #1 [c0431eb4] __handle_sysrq at c0254664
 #2 [c0431edc] handle_sysrq at c0254713
```

Let's next try peeking at the log messages generated by the crashed kernel. The `log` command reads the messages from the kernel `printk` ring buffer present on the dump file:

```
crash> log
...
BUG: soft lockup detected on CPU#0!

Pid: 2261, comm:               insmod
EIP: 0060:[<c010ec1b>] CPU: 0
EIP is at delay_pmtmr+0xb/0x20
EFLAGS: 00000246    Tainted: P       (2.6.16.16 #11)
EAX: 5caaa48c EBX: 00000001 ECX: 5caaa459 EDX: 00000012
ESI: 02e169c9 EDI: 00000000 EBP: 00000001 DS: 007b ES: 007b
CR0: 8005003b CR2: 08062017 CR3: 35e89000 CR4: 000006d0
 [<c01fede9>] __delay+0x9/0x10
 [<c0200089>] _raw_spin_lock+0xa9/0x150
 [<f893d00d>] mydrv_init+0xd/0xb2 [wqdrv]
 [<c0136565>] sys_init_module+0x175/0x17a2
 [<c015d834>] do_sync_read+0xc4/0x100
 [<c013ce4d>] audit_syscall_entry+0x13d/0x170
 [<c0105578>] do_syscall_trace+0x208/0x21a
 [<c0102f05>] syscall_call+0x7/0xb
SysRq : Trigger a crashdump
crash>
```

The log offers two useful pieces of debug information. First, it lets you know that a soft lockup was detected on the crashed kernel. The kernel detects soft lockups as follows: A kernel watchdog

thread runs once a second and touches a per-CPU timestamp variable. If the CPU sits in a tight loop, the watchdog thread cannot update this timestamp. An update check is carried out during timer interrupts using `softlockup_tick()` (defined in *kernel/softlockup.c*). If the watchdog timestamp is more than 10 seconds old, it concludes that a soft lockup has occurred and emits a kernel message to that effect.

Second, the log frowns upon `mydrv_init()+0xd` (or `0xf893d00`), so let's look at the disassembly of the surrounding code region:

```
crash> dis f893d000 5
dis: WARNING: f893d000: no associated kernel symbol found
0xf893d000:     mov    $0xf89f1208,%eax
0xf893d005:     sub    $0x8,%esp
0xf893d008:     call   0xc0342300 <_spin_lock>
0xf893d00d:     movl   $0xffffffff,0xf89f1214
0xf893d017:     movl   $0xffffffff,0xf89f1210
```

The return address in the stack is `0xf893d00d`, so the kernel is hanging inside the previous instruction, which is a call to `spin_lock()`. If you co-relate this with the earlier source snippet and look at it in the eye, you can see the error sequence, `spin_lock()/spin_lock_init()`, staring sorrowfully back at you. Fix the problem by swapping the sequence.

You can also use crash to peek at data structures of interest, but be aware that memory regions that were swapped out during the crash are not part of the dump. In the preceding example, you can examine `mydrv_wq` as follows:

```
crash> rd mydrv_wq 100
f892c200:  00000000 00000000 00000000 00000000   ................
...
f892c230:  2e636373 00000068 00000000 00000011   scc.h...........
```

Gdb is integrated with crash, so you can pass commands from crash to gdb for evaluation. For example, you can use gdb's `p` command to print data structures.

Looking at the Sources

Architecture-dependent portions of kexec reside in *arch/your-arch/kernel/machine_kexec.c* and *arch/your-arch/kernel/relocate_kernel.S.* The generic parts live in *kernel/kexec.c* (and *include/linux/kexec.h*). Peek inside *arch/your-arch/kernel/crash.c* and *arch/your-arch/kernel crash_dump.c* for the kdump implementation. *Documentation/kdump/kdump.txt* contains installation information.

Profiling

Profiling points you to those regions of code that burn more CPU cycles. Profilers help sense the presence of code bottlenecks and come in different flavors. The *OProfile* kernel profiler, included with the 2.6 kernel, uses hardware assist to gather profile data. The *gprof* application profiler, on the other hand, relies on compiler assist to collect profiling information.

Kernel Profiling with OProfile

OProfile samples data at regular intervals using hardware performance counters supported by many processors. The performance counters can be programmed to count events such as the

number of cache misses. On systems where the processor does not support performance counters, OProfile obtains limited information by collecting data during timer events.

OProfile consists of the following:

- A kernel layer that collects profiling information.[7] To enable OProfile in your kernel, enable `CONFIG_PROFILING`, `CONFIG_OPROFILE`, and `CONFIG_APIC` and recompile.
- The *oprofiled* daemon.
- A suite of post-profiling tools such as *opcontrol*, *opreport*, and *op_help* that help in detailed analysis of the collected data. These tools are included with several distributions; if your distribution doesn't have them, however, you can download precompiled binaries.

To illustrate the basics of kernel profiling, let's simulate a bottleneck in the filesystem layer and use OProfile to detect it. Our code area of interest is the portion of the filesystem layer that reads directories (function `vfs_readdir()` in *fs/readdir.c*)

First, use opcontrol to configure OProfile:

```
bash> opcontrol --setup --vmlinux=/path/to/kernelsources/vmlinux
                --event=GLOBAL_POWER_EVENTS:100000:1:1:1
```

The event specifier asks OProfile to collect samples during `GLOBAL_POWER_EVENTS` (time during which the processor is not stopped). The numerals adjacent to the event specifier denote the sampling count in clock cycles, unit mask filter, kernel-space counting, and user-space counting, respectively. If you want to sample `x` times every second and your processor is running at a frequency of

[7] *If you still use a 2.4 kernel, you need to patch your kernel sources with OProfile support.*

`cpu_speed` HZ, your sample count should approximately be `(cpu_speed/x)`. A larger count generates a finer profile but also results in more CPU overhead.

The events supported by OProfile depend on your processor:

```
bash> opcontrol -l  → List available events on a Pentium 4 CPU
GLOBAL_POWER_EVENTS: (counter: 0, 4)
       time during which processor is not stopped (min count: 3000)
BRANCH_RETIRED: (counter: 3, 7)
       retired branches (min count: 3000)
MISPRED_BRANCH_RETIRED: (counter: 3, 7)
       retired mispredicted branches (min count: 3000)
BSQ_CACHE_REFERENCE: (counter: 0, 4)
...
```

Next, start OProfile and run a benchmarking tool that stresses those parts of the kernel you would like to profile. Look at http://lbs.sourceforge.net/ for a list of benchmarking projects on Linux. For this example, let's exercise the *Virtual File System (VFS)* layer by recursively listing all files in the system:

```
bash> opcontrol -start    → Start data collection
bash> ls -lR /             → Stress test
bash> opcontrol -dump     → Save profiled data
```

Use opreport to look at the profiling results. The % column provides a measure of the function's load on the system:

```
bash> opreport -l /path/to/kernelsources/vmlinux

CPU: P4 / Xeon, speed 2992.9 MHz (estimated)
Counted GLOBAL_POWER_EVENTS events (time during which processor
is not stopped) with a unit mask of 0x01 (count cycles when processor is active)
count 100000
samples  %        symbol name
914506   24.2423  vgacon_scroll   → ls output printed to console
406619   10.7789  do_con_write
273023    7.2375  vgacon_cursor
206611    5.4770  __d_lookup
...
1380      0.0366  vfs_readdir     → Our routine of interest
...
1        2.7e-05  vma_prio_tree_next
```

Let's now simulate a bottleneck in the VFS code by introducing a 1-millisecond delay in `vfs_readdir()`. This is done in Listing 1.6.

LISTING 1.6 vfs_readdir() Defined in fs/read_dir.c

```
int vfs_readdir(struct file *file, filldir_t filler, void *buf)
{
  struct inode *inode = file->f_ dentry->d_inode;
  int res = -ENOTDIR;
```

```
+ /* Introduce a millisecond bottleneck
+    (HZ is set to 1000 on this system) */
+ unsigned long timeout = jiffies+1;
+ while (time_before(jiffies, timeout));
+ /* End of bottleneck */

  /* ... */
}
```

Compile the kernel with this change and recollect the profile. The new data looks like this:

```
bash> opreport -l /path/to/kernelsources/vmlinux

CPU: P4 / Xeon, speed 2993.08 MHz (estimated)
Counted GLOBAL_POWER_EVENTS events (time during which processor is not stopped) with a unit mask
of 0x01 (count cycles when processor is active)
count 100000
samples  %        symbol name
6178015  57.1640  vfs_readdir    → Our routine of interest
1065197   9.8561  vgacon_scroll  → ls output printed to console
479801    4.4395  do_con_write
...
```

As you can see, the bottleneck is clearly reflected in the profiled data. `vfs_readdir()` has now jumped to the top of the list!

You can use OProfile to obtain a lot more information. You can, for example, gather the percentage of data cache line misses. Caches are fast memory close to the processor. Fetches to cache are done in units of the processor cache line (32 bytes for Pentium 4). If the data you need to access is not already present in the cache (a cache miss), the processor has to fetch it from main memory, and this burns more CPU cycles. Subsequent accesses to that memory (and the surrounding bytes touched into the cache) will be faster until the corresponding cache line gets invalidated. You can configure OProfile to count the number of cache misses by profiling your kernel code for the `BSQ_CACHE_REFERENCE` event (for Pentium 4). You can then tune your code, possibly by realigning fields in data structures, to achieve better cache utilization:

```
bash> opcontrol --setup
                --event=BSQ_CACHE_REFERENCE:50000:0x100:1:1
                --vmlinux=/path/to/kernelsources/vmlinux
                                  → Unit mask 0x100 denotes an L2 cache miss
bash> opcontrol –start            → Start data collection
bash> ls -lR /                    → Stress
bash> opcontrol –dump             → Save profile
bash> opreport -l /path/to/kernelsources/vmlinux

CPU: P4 / Xeon, speed 2993.68 MHz (estimated)
Counted BSQ_CACHE_REFERENCE events (cache references seen by the bus unit) with a unit mask of
0x100 (read 2nd level cache miss) count 50000
samples  %        symbol name
73       29.6748  find_inode_fast
59       23.9837  __d_lookup
27       10.9756  inode_init_once
...
```

If you run OProfile on different kernel versions and look at the corresponding change logs, you might figure out reasons for code changes in different parts of the kernel.

You have only touched the surface of what can be accomplished using OProfile. For more information, visit http://oprofile.sourceforge.net/.

Application Profiling with Gprof

If you need to profile only an application process in isolation without profiling the kernel code that might get executed on its behalf, use *gprof* rather than OProfile. Gprof relies on additional code generated by the compiler to profile C, Pascal, or Fortran programs. Let's use gprof to profile the following code snippet:

```
main(int argc, char *argv[])
{
  int i;

  for (i=0; i<10; i++) {
    if (!do_task()) {       /* Perform task */
      do_error_handling();  /* Handle errors */
    }
  }
}
```

Use the `-pg` option to ask the compiler to include extra code that generates a call graph profile when the program runs. The `-g` option generates symbolic information:

```
bash> gcc -pg -g -o myprog myprog.c
bash> ./myprog
```

This produces *gmon.out*, which is a call graph of *myprog*. Run gprof to view the profile:

```
bash> gprof -p -b myprog

Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 65.17      2.75     2.75        2     1.38     1.38  do_error_handling
 34.83      4.22     1.47       10     0.15     0.15  do_task
```

This shows that the error path was hit twice during execution. You can tune the code to produce fewer traversals of the error path and rerun gprof to generate an updated profile.

Tracing

Tracing provides insight into behavioral problems that manifest during interactions between different code modules. A common way to obtain execution traces is by using `printk`s. Although `printk` is perhaps the most heavily used method for kernel debugging (there are more than 62,000 `printk()` statements in the 2.6.23 source tree), it is not sophisticated enough for high-volume tracing. *Linux Trace Toolkit (LTT)* is a powerful tool that lets you obtain complex system level traces with minimum overhead.

Linux Trace Toolkit

LTT extracts execution traces that are useful for postmortem analyses and is valuable in situations where it might not be possible to use a debugger. Unlike OProfile, which collects data by sampling events at regular intervals, LTT provides exact traces of events as and when they occur.

LTT consists of four components:

- A core module that provides trace services to the rest of the kernel. The collected traces are copied to a kernel buffer.
- Code that makes use of the trace services. These are inserted at points where you want to capture trace dumps.
- A trace daemon that pulls trace information from the kernel buffer to a permanent location in the filesystem.
- Utilities such as *tracereader* and *tracevisualizer* that interpret raw trace data and convert it into human-readable form. If you are developing code for an embedded device having no GUI support, you can transparently run these tools on another machine.

LTT is not part of the mainline kernel.[8] You might download LTT kernel patches, trace daemon, and user-space trace utilities from www.opersys.com/LTT.

Let's find out what LTT offers with the help of a simple example. Assume that you see data corruption when your application is reading from a device. You first want to figure out whether the

[8] *LTT was included as a release candidate in the 2.6.11-rc1-mm1 patch, downloadable from www.kernel.org.*

device is sending bad data or whether a kernel layer is introducing the corruption. To do that, dump data packets and data structures at the device driver level. Listing 1.7 initializes the LTT events that you plan to generate.

LISTING 1.7 Creating LTT Events

```
#include <linux/trace.h>

int data_packet, driver_data; /* Trace events */

/* Driver init */
static int __init mydriver_init(void)
{
  /* ... */

  /* Event to dump packets received from the device */
  data_packet = trace_create_event("data_pkt",
                                   NULL,
                                   CUSTOM_EVENT_FORMAT_TYPE_HEX,
                                   NULL);

  /* Event to dump a driver structure */
  driver_data = trace_create_event("dvr_data",
                                   NULL,
                                   CUSTOM_EVENT_FORMAT_TYPE_HEX,
                                   NULL);
```

```
  /* ... */

}
```

Next, let's add trace hooks to dump received packets and data structures when the driver reads data from the device. This is done in Listing 1.8 in the driver `read()` method.

LISTING 1.8 Obtaining Trace Dumps

```
struct mydriver_data driver_data; /* Private device structure */

/* Driver read() method */
ssize_t
mydriver_read(struct file *file, char *buf,
              size_t count, loff_t *ppos)
{
  char *buffer;

  /* Read numbytes bytes of data from the device into
     buffer */
  /* ... */

  /* Dump data Packet. If you see the problem only
     under certain conditions, say, when the packet length is
     greater than a value, use that as a filter */
  if (condition) {
    /* See Listing 1.7 for the definition of data_packet*/
```

```
    trace_raw_event(data_packet, numbytes, buffer);
  }

  /* Dump driver data structures */
  if (some_other_condition) {
    /* See Listing 1.7 for the definition of driver_data */
    trace_raw_event(driver_data, sizeof(driver_data), &driver_data);
  }

  /* ... */
}
```

Compile and run this code as part of the kernel or as a module. Remember to turn on trace support in the kernel by setting `CONFIG_TRACE` while configuring the kernel. The next step is to start the trace daemon:

```
bash> tracedaemon -ts60 /dev/tracer mylog.txt mylog.proc
```

/dev/tracer is the interface used by the trace daemon to access the trace buffer, `-ts60` asks the daemon to run for 60 seconds, *mylog.txt* is the file where you want to store the generated raw trace, and *mylog.proc* is where you want to save the system state obtained from procfs. Later versions of LTT use a mechanism called *relayfs* rather than the */dev/tracer* device for efficiently transferring data from the kernel trace buffer to user space.

Run your application that reads data from the device:

```
bash> ./application  → Trigger invocation of mydriver_read()
```

mylog.txt should now contain the requested trace data. The generated raw trace can be analyzed using the *tracevisualizer* tool. Choose the *Custom Events* option and search for `data_pkt` and `dvr_data` events. The output looks like this:

```
######################################################################
Event       Time SECS     MICROSEC    PID     Length  Description
######################################################################
data_pkt    1,110,563,008,742,457     0        27       12 43 AB AC 00 01 0D 56
data_pkt    1,110,563,008,743,151     0        27       01 D4 73 F1 0A CB DD 06
dvr_data    1,110,563,008,743,684     0        25       0D EF 97 1A 3D 4C
...
```

The last column holds the trace data. The timestamp shows the instant when the trace was collected. If the data looks corrupt, the device could be sending bad data. Otherwise, the problem must be in a higher kernel layer and can be further isolated by obtaining traces from a different point in the data-flow path.

The next generation of LTT called LTTng is available for download from http://ltt.polymtl.ca/. This project also includes a post-trace analyzer called *Linux Trace Toolkit Viewer (LTTV)*.

If your need is only to perform limited tracing of a user application, you can use the *strace* utility rather than LTT. Strace uses the *ptrace* support in the kernel to intercept system calls. It prints out a list of system calls made by your application, along with the corresponding arguments and return values.

Debugging Embedded Linux

From a device driver perspective, embedded software developers often face interfaces not commonly found on conventional computers. Figure 1.3 shows a hypothetical embedded device that could be a handheld, smart phone, *point-of-sale (POS)* terminal, kiosk, navigation system, gaming device, telemetry gadget on an automobile dashboard, IP phone, music player, digital set-top box, or even a pacemaker programmer. The device is built around a System on Chip (SoC) and has some combination of flash memory, SDRAM, Liquid Crystal Display (LCD), serial ports, touch screen, Universal Serial Bus (USB), audio codec, connectivity, Secure Digital/Multimedia Card (SD/MMC) controller, Compact Flash, Inter-Integrated Circuit (I^2C) devices, Serial Peripheral Interface (SPI) devices, biometrics, smart card interfaces, keypad, switches, and electronics specific to the industry domain. Modifying and debugging drivers for some of these devices can be tougher than usual: NAND flash drivers have to handle problems such as bad blocks and failed bits, unlike standard hard disk drivers. Flash-based filesystems such as Journalling Flash File System (JFFS) are more complex to debug than EXT2 or EXT3 filesystems. A USB On-The-Go (OTG) driver is more involved than a conventional USB Open Host Controller Interface (OHCI) driver. The SPI sub-system in the kernel is not as mature as the serial layer. Moreover, the industry domain using the embedded device might impose specific requirements such as deterministic response times or fast boot.

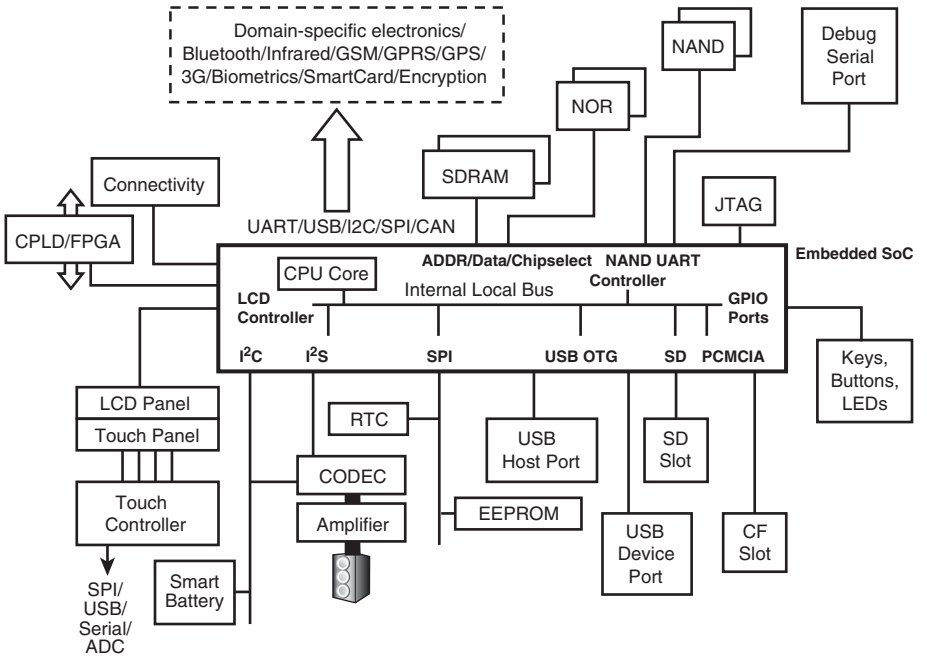


FIGURE 1.3
Block diagram of a hypothetical embedded device

Let's collect some pointers on debugging device drivers for I/O interfaces commonly found on embedded hardware. To do this, let's take a clockwise tour around the embedded SoC shown in Figure 1.3, starting with the NOR flash. Before we end this section, let's also learn some board-level debugging skills.

Flash Memory

Embedded devices usually boot from flash memory and have filesystem data resident on flash-based storage. Many devices use a small NOR flash component for the former and a NAND flash

part for the latter. NOR memory, thus, holds the bootloader and the base kernel, whereas NAND storage contains filesystem partitions and device driver modules.

Flash drivers are supported by the kernel's Memory Technology Devices (MTD) subsystem. If you use an MTD-supported chip, you need to write only an MTD map driver to suitably partition the flash to hold the bootloader, kernel, and filesystem. The MTD subsystem resides in the *drivers/mtd/* directory in the kernel source tree.

To debug flash-related problems, enable `CONFIG_MTD_DEBUG` (*Device Drivers→Memory Technology Devices→Debugging*) during kernel configuration. You can further tune the debug verbosity level to between 0 and 3.

The Linux MTD project page www.linux-mtd.infradead.org/ has FAQs, various pieces of documentation, and a *Linux MTD JFFS HOWTO* that provides insights into JFFS2 design. The linux-mtd mailing list is the place to discuss questions related to MTD device drivers. Look at http://lists.infradead.org/pipermail/linux-mtd/ for the mailing list archives.

Serial Port

The Universal Asynchronous Receiver/Transmitter (UART) chip is responsible for serial communication and is an interface you are likely to find on all SoCs. UARTs are considered basic hardware, so the kernel contains UART drivers for all SoCs on which it runs. On embedded devices, UARTs are used to interface the processor with debug serial ports, modems, touch controllers, cellular chipsets, Bluetooth silicon, Global Positioning System (GPS) units, telemetry electronics, and so on.

In the Linux source tree, the *drivers/serial/* directory contains the source code of the serial subsystem.

If the UART is meant to be your system console, debugging it isn't an easy job because you can't invoke `printk()` from inside the driver. If you have another console device and an associated working driver, you can print your debug messages to that device, however.

PCMCIA and Compact Flash

PCMCIA (Personal Computer Memory Card International Association) or Compact Flash (CF) slots are common add-ons to embedded devices. The advantage of WiFi-enabling an embedded device using a CF card is that you don't need to re-spin the board if the WiFi controller goes end of life. Also, because diverse technologies are available in the PCMCIA/CF form factor, you've the freedom to change the connectivity mode from WiFi to another technology such as Bluetooth later. The disadvantage of such a scheme is that even with mechanical retaining, sockets are inherently unreliable. There is the possibility of the card coming loose due to shock and vibe, and resulting intermittent connections.

The kernel's PCMCIA/CF subsystem lives in the *drivers/pcmcia/* directory.

To effectively debug PCMCIA/CF client drivers, you need to see debug messages emitted by the PCMCIA core. For this, enable `CONFIG_PCMCIA_DEBUG` (*Bus options→PCCARD support→Enable PCCARD debugging*) during kernel configuration. Verbosity levels of the debug output can be controlled either via the `pcmcia_core.pc_debug` kernel command-line argument or using the `pc_debug` module insertion parameter.

Information about PC Card client drivers is available in the process filesystem entry, */proc/bus/pccard/drivers*. Look at */sys/bus/pcmcia/devices/** for card-specific information such as manufacturer and card IDs. Take a look inside */proc/bus/pci/* to know more about your PCMCIA host controller if your system uses a PCI-to-PCMCIA bridge. */proc/interrupts* lists IRQs active on your system, including those used by the PCMCIA layer.

You can find a mailing list dedicated to Linux-PCMCIA at http://lists.infradead.org/mailman/listinfo/linux-pcmcia.

Secure Digital Media

Many embedded processors include controllers that communicate with SD and MMC media. SD/MMC storage is built using NAND flash memory. Like CF cards, SD/MMC cards add several gigabytes of memory to your device. They also offer an easy memory upgrade path, because the available density of SD/MMC cards is constantly increasing.

The kernel contains an SD/MMC subsystem in *drivers/mmc/*.

The *hdparm* utility elicits various PATA/SATA disk parameters from the underlying kernel block driver. To benchmark disk read speeds on a SATA drive, for example, do this:

```
bash> hdparm -T -t /dev/sda
/dev/sda:
 Timing cached reads: 2564 MB in 2.00 seconds = 1283.57 MB/sec
 Timing buffered disk reads: 132 MB in  3.03 seconds = 43.61 MB/sec
```

For the full capabilities of hdparm, read the man pages.

Self-Monitoring, Analysis and Reporting Technology (SMART) is a system built into many modern ATA and SCSI disks to monitor failures and perform self-tests. A user space daemon named *smartd* collects the information gathered by SMART-capable disks with the help of the underlying device driver. Look at the man pages of smartd, smartctl, and smartd.conf to learn how to obtain health status from SMART-enabled disks.

If your distribution doesn't pre-package hdparm and SMART tools, you can download them from http://sourceforge.net/projects/hdparm/ and http://sourceforge.net/projects/smartmontools/, respectively.

Files under */proc/ide/* contain information about IDE disk drives on your system. To obtain the geometry of the first IDE disk, look at the contents of */proc/ide/ide0/hda/geometry*. Information pertaining to SCSI devices is available under */proc/scsi/*. You can gather disk partition information from */proc/partitions*.

The *sysfs* directory of interest for IDE devices is */sys/bus/ide/*, whereas for SCSI is */sys/bus/scsi/*. Additionally, each block device active on the system owns a subdirectory under */sys/block/*, which contains associated request queue parameters, constituent partition details, and state information.

Some kernel configuration options are available that trigger emission of debug output from the block subsystem. `CONFIG_BLK_DEV_IO_TRACE` provides the ability to trace the block layer. `CONFIG_SCSI_CONSTANTS` and `CONFIG_SCSI_LOGGING` turn on SCSI error reporting and logging, respectively.

The linux-ide mailing list is the forum to discuss questions related to the Linux-IDE subsystem. Subscribe to the linux-scsi mailing list and browse through its archives for discussions pertaining to the Linux-SCSI subsystem.

Universal Serial Bus

Legacy computers support the USB host mode by which you can communicate with most classes of USB devices. Embedded systems frequently also require support for the USB device mode, wherein the system itself functions as a USB device and plugs into other host computers.

Many modern embedded controllers support USB OTG that lets your device work either as a USB host or as a USB device. It enables you, for example, to connect a USB pen drive to your embedded device. It also enables your embedded device to itself function as a USB pen drive by exporting part of its local storage for external access. The Linux USB subsystem (residing in *drivers/usb/*) offers drivers for USB OTG. For hardware that's not compatible with OTG, the USB Gadget project (implemented in *drivers/usb/gadget/*), brings USB device capability.

A USB bus analyzer magnifies the goings-on in the bus and is useful for debugging low-level problems. If you can't get hold of an analyzer, you might make do with the kernel's soft USB tracer *usbmon*. This tool captures traffic between USB host controllers and devices. To collect a trace, read from the *debugfs*[9] file */sys/kernel/debug/usbmon/Xt*, where *X* is the bus number to which your device is connected.

For example, consider a USB disk connected to a PC. From the associated "T:" line in */proc/bus/usb/devices*, you can see that the drive is attached to bus 1:

```
T:  Bus=01 Lev=01 Prnt=01 Port=03 Cnt=01 Dev#=  2 Spd=480 MxCh= 0
```

[9] *An in-memory filesystem to export kernel debug data to user space.*

Ensure that you have enabled debugfs (`CONFIG_DEBUG_FS`) and usbmon (`CONFIG_USB_MON`) support in your kernel. This is a snapshot of usbmon output while copying a file from the disk:

```
bash> mount -t debugfs none_debugs /sys/kernel/debug/
bash> cat /sys/kernel/debug/usbmon/1u
...
ee6a5c40 3718782540 S Bi:1:002:1 -115 20480 <
ee6a5cc0 3718782567 S Bi:1:002:1 -115 65536 <
ee6a5d40 3718782595 S Bi:1:002:1 -115 36864 <
ee6a5c40 3718788189 C Bi:1:002:1 0 20480 = 0f846801 118498f\
15c60500 01680106 5e846801 608498fe 6f280087 68000000
ee6a5cc0 3718800994 C Bi:1:002:1 0 65536 = 118498fe 15c60500\
01680106 5e846801 608498fe 6f280087 68000000 00884800
ee6a5d40 3718801001 C Bi:1:002:1 0 36864 = 13608498 fe4f4a01\
00514a01 006f2800 87680000 00008848 00000100 b7f00100
...
```

Each output line starts with the URB address, followed by an event timestamp. An *S* in the next column indicates URB submission, whereas a *C* announces a callback. The following field has the format `URBType:Bus#:DeviceAddress:Endpoint#`. In the preceding output, a `URBType` of `Bi` stands for a bulk URB in the `IN` direction. After this, usbmon dumps the URB status, data length, a data tag (= or < in the preceding output), and the data words (if the tag is =). The last three lines in the preceding output are callbacks associated with bulk URBs submitted in earlier lines. You can match the callbacks with the related submissions using the URB addresses. *Documentation/usb/usbmon.txt* details usbmon syntax and contains example code to parse the output into human readable form.

If you turn on *Device Drivers → USB support → USB verbose debug messages* during kernel configuration, the kernel will emit the contents of all `dev_dbg()` statements present in the USB subsystem.

You can glean device and bus specific information from the USB filesystem (*usbfs*) node */proc/bus/usb/devices*. usbfs also lets you implement USB device drivers in user space. Even when the final destination of your USB driver is inside the kernel, starting with a user space driver can ease debugging and testing.

The linux-usb-devel mailing list is the forum to discuss questions related to USB device drivers. Visit https://lists.sourceforge.net/lists/listinfo/linux-usb-devel for subscription and archive retrieval information. Read www.linux-usb.org/usbtest for ideas on USB testing.

The home page of the Linux-USB project is www.linux-usb.org/. You can download the USB 2.0 specification, OTG supplement, and other related standards from www.usb.org/developers/docs/.

Real Time Clock

Many embedded SoCs include RTC support to keep track of wall time, but some rely on an external RTC chip. Unlike x86-based computers where the RTC is part of the South Bridge chipset, embedded controllers commonly interface with external RTCs via slow serial buses such as I^2C or SPI. You can drive such RTCs by writing client drivers that use the services of the I^2C or SPI core layers. The former drivers can be found under *drivers/i2c/*, and the latter under *drivers/spi/*.

To collect I^2C-specific debugging messages, turn on a relevant combination of I^2C Core debugging messages, I^2C Algorithm debugging messages, I^2C Bus debugging messages, and I^2C Chip debugging messages under *Device Drivers→I^2C* support in the kernel configuration menu. Similarly, for SPI debugging, turn on Debug Support for SPI drivers under *Device Drivers→SPI* support.

To understand the flow of I²C packets on the bus, connect an I²C bus analyzer to your board. The lm-sensors package contains a tool called i2cdump that dumps register contents of devices on the I²C bus.

You can find a mailing list dedicated to Linux I²C at http://lists.lm-sensors.org/mailman/listinfo/i2c.

Audio

An audio codec converts digital audio data to analog sound signals for playback via speakers and performs the reverse operation for recording through a microphone. The codec's connection with the CPU depends on the digital audio interface supported by the embedded controller. The usual way to communicate with a codec is via the Audio Codec'97 (AC'97) bus or the Inter-IC Sound (I²S) bus for data and I²C for the control interface.

Advanced Linux Sound Architecture (*ALSA*) is the sound framework of choice in the 2.6 kernel. ALSA obsoletes *Open Sound System* (*OSS*), which was the sound subsystem in 2.4 kernels. The kernel's sound core and audio bus drivers stay inside the top-level *sound/* directory in the kernel source tree.

You may turn on options under *Device Drivers→Sound→Advanced Linux Sound Architecture* in the kernel configuration menu to include ALSA debug code (`CONFIG_SND_DEBUG`), verbose `printk()` messages (`CONFIG_SND_VERBOSE_PRINTK`), and verbose procfs content (`CONFIG_SND_VERBOSE_PROCFS`).

Procfs information pertaining to ALSA drivers resides in */proc/asound/*. Look inside */sys/class/sound/* for the device model information associated with each sound-class device.

If you think you have found a bug in an ALSA driver, post it to the alsa-devel mailing list (http://mailman.alsa-project.org/mailman/listinfo/alsa-devel). The linux-audio-dev mailing list (http://music.columbia.edu/mailman/listinfo/linux-audio-dev/), also called the *Linux Audio Developers* (*LAD*) list, discusses questions related to the Linux-Sound architecture and audio applications.

Touch Screen

Touch is the primary input mechanism on several embedded devices. Many handhelds offer soft touch keyboards for data entry.

The kernel offers an *input* subsystem for the uniform handling of functionally similar input devices even if they are physically different. For example, all mice, whether they are PS/2, USB, or Bluetooth, are treated alike. The kernel also provides an *event* abstraction for dispatching input reports to user applications. Applications such as X Windows work seamlessly over the event interfaces exported by the input subsystem.

If your touch controller driver conforms to the input event abstraction, it's straightforward to tie it with a *graphical user interface* (*GUI*).

You can use the *evbug* module as a debugging aid if you're developing an input driver. It dumps the (*type*, *code*, *value*) tuple corresponding to events generated by the input subsystem. Figure 1.4 contains data captured by evbug while operating some input devices.

FIGURE 1.4
Evbug output

```
/* Touchpad Movement */
evbug.c Event. Dev: isa0060/serio1/input0: Type: 3, Code: 28, Value: 0
evbug.c Event. Dev: isa0060/serio1/input0: Type: 1, Code: 325, Value: 0
evbug.c Event. Dev: isa0060/serio1/input0: Type: 0, Code: 0, Value: 0

/* Trackpoint Movement */
evbug.c Event. Dev: synaptics-pt/serio0/input0: Type: 2, Code: 0, Value: -1
evbug.c Event. Dev: synaptics-pt/serio0/input0: Type: 2, Code: 1, Value: -2
evbug.c Event. Dev: synaptics-pt/serio0/input0: Type: 0, Code: 0, Value: 0

/* USB Mouse Movement */
evbug.c Event. Dev: usb-0000:00:1d.1-2/input0: Type: 2, Code: 1, Value: -1
evbug.c Event. Dev: usb-0000:00:1d.1-2/input0: Type: 0, Code: 0, Value: 0
evbug.c Event. Dev: usb-0000:00:1d.1-2/input0: Type: 2, Code: 0, Value: 1
evbug.c Event. Dev: usb-0000:00:1d.1-2/input0: Type: 0, Code: 0, Value: 0

/* PS/2 Keyboard keypress 'a' */
evbug.c Event. Dev: isa0060/serio0/input0: Type: 4, Code: 4, Value: 30
evbug.c Event. Dev: isa0060/serio0/input0: Type: 1, Code: 30, Value: 0
evbug.c Event. Dev: isa0060/serio0/input0: Type: 0, Code: 0, Value: 0

/* USB keyboard keypress 'a' */
evbug.c Event. Dev: usb-0000:00:1d.1-1/input0: Type: 1, Code: 30, Value: 1
evbug.c Event. Dev: usb-0000:00:1d.1-1/input0: Type: 0, Code: 0, Value: 0
evbug.c Event. Dev: usb-0000:00:1d.1-2/input0: Type: 1, Code: 30, Value: 0
evbug.c Event. Dev: usb-0000:00:1d.1-2/input0: Type: 0, Code: 0, Value: 0
```

To make sense of the dump in Figure 1.4, remember that touchpads generate absolute coordinates (`EV_ABS`) or event type `0x03`, trackpoints produce relative coordinates (`EV_REL`) or event type `0x02`, and keyboards emit key events (`EV_KEY`) or event type `0x01`. Event type `0x0` corresponds to an invocation of `input_sync()`, which does the following:

```
input_event(dev, EV_SYN, SYN_REPORT, 0);
```

This translates to a (*type*, *code*, *value*) tuple of (`0x0`, `0x0`, `0x0`) and completes each input event.

Video

Some embedded systems are headless, but many have associated displays. A suitably oriented (landscape or portrait) LCD panel is connected to the video controller that's part of the embedded SoC.

The kernel's frame buffer interface insulates applications from display hardware, so porting a compliant GUI to your device is easy if your display driver conforms to the kernel's *frame buffer* abstraction. The frame buffer core layer and low-level frame buffer drivers reside in the *drivers/video/* directory.

The virtual frame buffer driver, enabled by setting `CONFIG_FB_VIRTUAL` in the configuration menu, operates over a pseudo graphics adapter. You can use this driver's assistance to debug the frame buffer subsystem.

Some frame buffer drivers, such as *intelfb*, offer an additional configuration option that you can enable to generate driver-specific debug information.

To discuss issues related to frame buffer drivers, subscribe to the linux-fbdev-devel mailing list, https://lists.sourceforge.net/lists/listinfo/linux-fbdev-devel/.

Bluetooth

Connectivity injects intelligence, so few embedded devices have no communication capability. Popular networking technologies found on embedded devices include Bluetooth, Infrared, WiFi, cellular modems, and radio communication.

Bluetooth eliminates cables and opens a flood gate of novel applications. The *BlueZ* Bluetooth implementation is part of the mainline kernel and is the official Linux Bluetooth stack. Look inside *drivers/bluetooth/* for low-level BlueZ drivers.

There are two BlueZ tools useful for debugging:

1. *hcidump* taps HCI packets flowing back and forth and parses them into human-readable form. Here's an example dump while a device inquiry is in progress:

```
bash> hcidump -i hci0
HCIDump - HCI packet analyzer ver 1.11
device: hci0 snap_len: 1028 filter: 0xffffffff
   HCI Command: Inquiry (0x01¦0x0001) plen 5
   HCI Event: Command Status (0x0f) plen 4
   HCI Event: Inquiry Result (0x02) plen 15
   ...
HCI Event: Inquiry Complete (0x01) plen 1 < HCI Command:
   Remote Name Request (0x01¦0x0019) plen 10
   ...
```

2. The virtual HCI driver (*hci_vhci.ko*) emulates a Bluetooth interface if you do not have actual hardware.

Board Rework

Navigating board schematics and datasheets is an important debugging skill you need while bringing up the bootloader or kernel on embedded hardware. Understanding your board's *placement*

plot, which is a file that shows the position of chips on your board, is a big help when you debug a potential hardware problem using an oscilloscope, or when you need to perform minor board rework. *Reference designators* (such as U10 and U11 in Figure 1.5) associate each chip in the schematic with the placement plot. *Printed Circuit Boards* (*PCBs*) are usually clothed with *silk screens* that print the reference designator near each chip.

Consider this fictitious scenario in which USB enumeration doesn't occur on your board under test. The USB hub driver detects device insertions but cannot assign endpoint addresses. A close look at the schematics reveals that the connections originating from the `SPEED` and `MODE` pins of the USB transceiver have been interchanged by mistake. An examination of the placement plot identifies the location of the transceiver on the PCB. Matching the transceiver's reference designator on the placement plot with the silk screen on the PCB pinpoints the places where you have to solder "yellow wires" to repair the faulty connections.

A multimeter and an oscilloscope are worthy additions to your embedded debugging toolkit. As an illustration, let's consider an example situation involving the I^2C RTC shown in Figure 1.5. Consider this scenario: You have written an I^2C client driver for this RTC chip. However, when you run your driver on the board, it renders the system unbootable. Neither does the bootloader come up when you reset the board, nor does your JTAG debugger connect to the target. To understand possible causes of this seemingly fatal error, let's take a closer look at the connection diagram. Because both the RTC and the CPU need an external clock, the board supplies it using a single 32 KHz crystal. This 32 KHz clock needs to be buffered, however. The RTC buffers the clock for its use and makes it available on an output pin for free. This pin `CLK_OUT`, feeds the clock to the processor. Connect an oscilloscope (or a multimeter that can measure frequency) between `CLK_OUT` and ground to verify the processor clock frequency. As you can see in Figure 1.5, the scope reads 1 KHz rather than the expected 32 KHz! What could be wrong here?

The RTC control register contains bits that choose the frequency of `CLK_OUT`. While probing the chip, the driver has erroneously initialized these bits to generate 1 KHz on `CLK_OUT`. RTC registers are nonvolatile because of the battery backup, so the control register holds this bad value across reboots. The resulting skewed clock is sufficient to render the system unbootable. Disconnect the RTC's backup battery, drain the registers, reconnect the battery, verify using the scope that the 32 KHz clock is restored on `CLK_OUT`, fix your driver code, and start afresh!

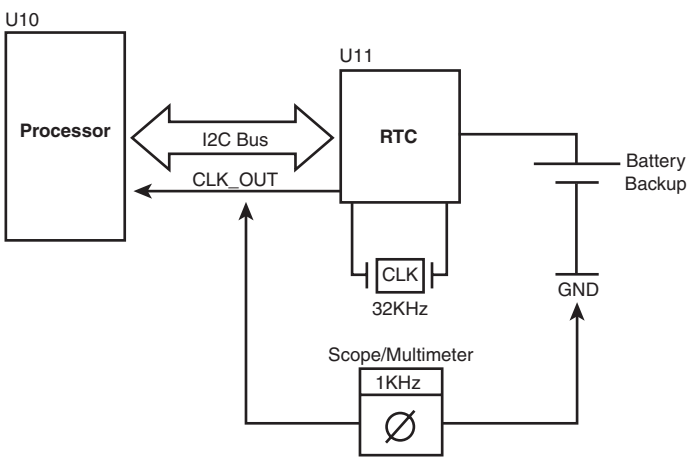


FIGURE 1.5
Debugging an I^2C RTC on an embedded system

Test Equipment

It goes without saying that you need the full complement of relevant test equipment for device driver debugging. If you test a modem interface in a digital-only laboratory environment for example, you will be well served by a phone simulator. If a high-speed serial driver is manifesting parity errors, an oscilloscope can aid your problem analyses. If you are writing an I/O device driver,

it helps if you have the associated bus analyzer. If you are writing a network driver, the corresponding protocol line sniffer eases your debugging effort.

Debugging Network Throughput

To obtain optimal network throughput, you need to design your *network interface card (NIC)* driver for high performance. Additionally, you need an in-depth understanding of the network protocol that your driver ferries. To get a flavor of how to debug network throughput problems, let's examine some device driver design issues and protocol implementation characteristics that can affect the horse power of your NIC.

Driver Performance

Let's first take a look at some driver design issues that can affect your NIC's performance:

- Minimizing the number of instructions in the main data path is a key criterion while designing drivers for fast NICs. Consider a 1 Gbps Ethernet adapter with 1 MB of onboard memory. At line rate, the card memory can hold up to 8 milliseconds of received data. This directly translates to the maximum allowable instruction path length. Within this path length, incoming packets have to be reassembled, DMA-ed to memory, processed by the driver, protected from concurrent access, and delivered to higher layer protocols.
- During programmed I/O (PIO), data travels all the way from the device to the CPU before it gets written to memory. Moreover, the CPU is interrupted whenever the device needs to transfer data, and this contributes to latencies and context switch delays. DMAs do not suffer from these bottlenecks but can turn out to be more expensive than PIOs if the data to be transferred

is less than a threshold. This is because small DMAs have high relative overheads for building descriptors and flushing corresponding processor cache lines for data coherency. A performance-sensitive device driver might use PIO for small packets and DMA for larger ones, after experimentally determining the threshold.

- For PCI network cards having DMA mastering capability, you need to determine the optimal DMA burst size, which is the time for which the card controls the bus at one stretch. If the card bursts for a long duration, it might hog the bus and prevent the processor from keeping up with data DMA-ed previously. PCI drivers program the burst size via a register in the PCI configuration space. Normally the NIC's burst size is programmed to be the same as the cache line size of the processor, which is the number of bytes that the processor reads from system memory each time there is a cache miss. In practice, however, you might need to connect a bus analyzer to determine the beneficial burst duration because factors such as the presence of a split bus (multiple bus types like ISA and PCI) on your system can influence the optimal value.
- Many high-speed NICs offer the capability to offload the CPU-intensive computation of TCP checksums from the protocol stack. Some support DMA scatter-gather. The driver needs to leverage these capabilities to achieve the maximum practical bandwidth that the underlying network yields.
- Sometimes, a driver optimization might create unexpected speed bumps if it's not sensitive to the implementation details of higher protocols. Consider an NFS-mounted filesystem on a computer equipped with a high-speed NIC. Assume that the NIC driver takes only occasional transmit complete interrupts to minimize latencies but that the NFS server implementation uses freeing of its transmit buffers as a flow-control mechanism. Because the driver frees NFS transmit buffers only during the sparsely generated transmit complete interrupts, file copies over NFS crawl, even as Internet downloads zip along yielding maximum throughput.

Protocol Performance

Let's now dig into some protocol-specific characteristics that can boost or hurt network throughput:

- TCP window size can impact throughput. The window size provides a measure of the amount of data that can be transmitted before receiving an acknowledgment. For fast NICs, a small window size might result in TCP sitting idle, waiting for acknowledgments of packets already transmitted. Even with a large window size, a small number of lost TCP packets can affect performance because lost frames can use up the window at line speeds. In the case of UDP, the window size is not relevant because it does not support acknowledgments. However, a small packet loss can spiral into a big rate drop due to the absence of flow-control mechanisms.
- As the block size of application data written to TCP sockets increases, the number of buffers copied from user space to kernel space decreases. This lowers the demand on processor utilization and is good for performance. If the block size crosses the MTU corresponding to the network protocol, however, processor cycles get wasted on fragmentation. The desirable block size is thus the outgoing interface MTU, or the largest packet that can be sent without fragmentation through an IP path if Path MTU discovery mechanisms are in operation. While running IP over ATM, for example, because the ATM adaptation layer has a 64K MTU, there is virtually no upper bound on block size. (RFC 1626 defaults this to 9180.) If you run IP over ATM LANE, however, the block size should mirror the MTU size of the respective LAN technology being emulated. It should thus be 1500 for standard Ethernet, 8000 for jumbo Gigabit Ethernet, and 18 K for 16 Mbps Token Ring.

Several tools are available to benchmark network performance. *Netperf*, available for free from www.netperf.org/, can set up complex TCP/UDP connection scenarios. You can use scripts to

control characteristics such as protocol parameters, number of simultaneous sessions, and size of data blocks. Benchmarking is accomplished by comparing the resulting throughput with the maximum practical bandwidth that the networking technology yields. For example, a 155 Mbps ATM adapter produces a maximum IP throughput of 135 Mbps, taking into account the ATM cell header size, overheads due to the *ATM Adaptation Layer (AAL)*, and the occasional maintenance cells sent by the physical *Synchronous Optical Networking (SONET)* layer.

Linux Test Project

Linux Test Project (LTP), hosted at http://ltp.sourceforge.net/, is a suite consisting of approximately 3,000 tests designed to exercise different parts of the kernel. Most tests run without user intervention. Others, such as networking and storage media tests, need some manual configuration.

Download the source tar ball from the LTP home page, run *make*, and invoke the wrapper script *runltp* from the root of the source tree to start the tests. To capture the results in *logfile* in the *results/* directory, do this:

```
bash> runltp -p -l logfile
```

Some errors generated by LTP are "expected." The LTP website documents the list of expected errors for various kernel versions. Also in the website is an interesting analysis of LTP's code coverage (overall coverage, lines in path, and distinct lines hit) for a few kernel versions, split across directories in the kernel tree.

User Mode Linux

User Mode Linux (UML), hosted at http://user-mode-linux.sourceforge.net/, lets you debug the kernel without "oops"ing the machine. To accomplish this, an instance of the kernel (called the *guest* kernel) runs as a user mode process over the running kernel (called the *host* kernel).

UML has diverse users. It can function as an environment for testing kernel and application code, a vehicle to experiment with unstable kernels, a secure pseudo computer for hosting services such as web servers, or a tool to learn Linux internals. UML is more useful for debugging hardware-independent portions of the kernel than for device driver debugging.

Diagnostic Tools

The *sysfsutils* package helps you navigate the voluminous amount of data present inside sysfs. This, and other Linux diagnostic tools such as *sysdiag* and *lsvpd*, can be downloaded from http://linux-diag.sourceforge.net/.

Kernel Hacking Config Options

Several options exist under *Kernel hacking* in the kernel configuration menu that can emit valuable debug information. If you enable an option, corresponding debug code compile when you build the kernel.[10] Here are a few examples:

[10] *Some kernel hacking options are architecture-dependent.*

1. *Show Timing information on printks* (`CONFIG_PRINTK_TIME`) adds timing instrumentation to `printk()` output, so you can use `printks` as checkpoints for measuring execution times and identifying slow code regions.
2. Using freed memory results in memory poisoning. *Debug slab memory allocations* (`CONFIG_DEBUG_SLAB`) helps you detect such problems.
3. *Spinlock and rw-lock debugging: basic checks* (`CONFIG_DEBUG_SPINLOCK`) finds lock-related problems such as uninitialized spinlock usage and helps catch code that is not SMP-safe.
4. You have already worked with *Magic SysRq key* (`CONFIG_MAGIC_SYSRQ`) when you learned to use kdump. If you turn this on, you have some avenues left even if the kernel crashes during debugging. For example, pressing Alt-Sysrq-t produces a dump of current tasks, whereas Alt-Sysrq-p prints the contents of processor registers.
5. *Detect Soft Lockups* (`CONFIG_DETECT_SOFTLOCKUP`) utilizes the services of a watchdog to detect tight loops in kernel code that last for more than 10 seconds. We looked at this when we analyzed a kernel hang using kdump. Note that hardware lockups cannot be found this way. For that, use the services of a Non-Maskable Interrupt (NMI)-watchdog if your platform supports it.
6. If you enable `CONFIG_DEBUG_SLAB`, `CONFIG_DEBUG_HIMEM`, or `CONFIG_DEBUG_PAGE_ALLOC` while configuring your kernel, additional error-checking code is compiled that help debug problems related to memory management.
7. *Check for stack overflows* (`CONFIG_DEBUG_STACKOVERFLOW`) adds code to emit warnings if the available stack space falls below a threshold. *Stack utilization instrumentation* (`CONFIG_DEBUG_STACK_USAGE`) adds stack space instrumentation to the magic Sysrq key output. Another related option, `CONFIG_4KSTACKS`, lets you set the kernel stack size to 4 KB rather than 8 KB.

8. *Verbose BUG() reporting* (`CONFIG_DEBUG_BUGVERBOSE`) produces extra debug information when any kernel code invokes `BUG()`, assuming that you have `-CONFIG_BUG` turned on during kernel configuration.

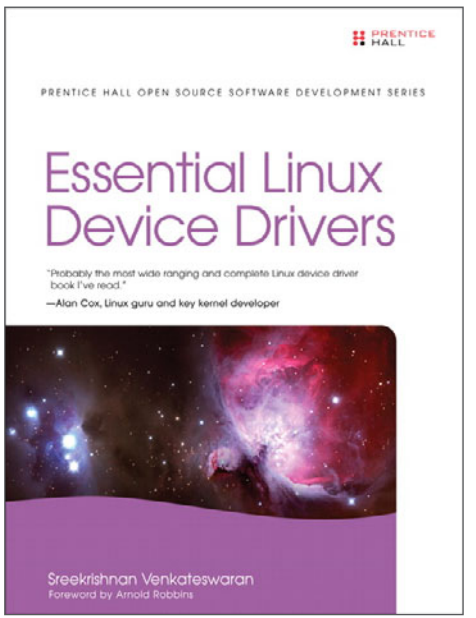
Some debug options live outside the *Kernel hacking* submenu, too. For example, we enabled `CONFIG_KALLSYMS` to debug an "oops" message using gdb and to kprobe a kernel module. This option lives under *General setup* → *Configure Standard Kernel Features (for small systems)* in the configuration menu.

Kernel hacking options result in overhead and increased footprint, so don't leave them on in production-level kernels.

The accessible, comprehensive tutorial on writing Linux device drivers for the beginning Linux driver writer

- Covers essential information on kernel concepts, driver concepts, wireless Linux concepts, and embedded Linux concepts—information never before available in one book.
- Includes several types of drivers not covered elsewhere such as WiFi, USB Serial Drivers, Sound drivers and such
- Describes the driver development lifecycle, including detailed debugging techniques

This book is explicitly targeted at the beginner-to-intermediate-level reader. Unlike *Linux Device Drivers*, Third Edition (ORA), it carefully introduces each new technology prior to embarking on a description of its respective kernel implementation. It presents only essential information by cutting-down/eliminating sections that are usually of interest only to advanced readers or sophisticated driver writers (for example an in-depth look into page tables and VM). And it does not dwell on details that tend to change rapidly as the kernel evolves (such as kernel API listings or digging deep into the semantics of kernel data structures). In short, it covers the essential information that new driver writers need to understand to be productive.



Sreekrishnan Venkateswaran

ISBN-13: 9780132396554

informIT For more information and to read sample material, please visit informit.com.

Safari Books Online Titles are also available at safari.informit.com.

PRENTICE HALL